Live RGB-D Camera Tracking for Television Production Studios

Tommy Tykkala, Andrew I. Comport, Joni-Kristian Kamarainen, Hannu Hartikainen

Live RGB-D Camera Tracking for Television Production Studios

Tommi Tykkälä[1], Andrew I. Comport[2], Joni-Kristian Kämäräinen[3], and Hannu Hartikainen[4]

[1]Machine Vision and Pattern Recognition Laboratory, Lappeenranta University of Technology, Kouvola Unit, Finland
[2] CNRS-I3S/University of Nice Sophia-Antipolis, France
[3] Department of Signal Processing, Tampere University of Technology, Finland
[4] Department of Media Technology, Aalto University, Finland

Abstract

In this work, a real-time image-based camera tracker is designed for live television production studios. The major concern is to decrease camera tracking expenses by an affordable vision-based approach. First, a dense keyframe model of the static studio scene is generated using image-based dense tracking and bundle adjustment. Online camera tracking is then defined as registration problem between the current RGB-D measurement and the nearest keyframe. With accurate keyframe poses, our camera tracking becomes virtually driftless. The static model is also used to avoid moving actors in the scene. Processing dense RGB-D measurements requires special attention when aiming for real-time performance at 30Hz. We derive a real-time tracker from our cost function for a low-end GPU. The system requires merely a RGB-D sensor, laptop and a low-end GPU. Camera tracking properties are compared with KinectFusion. Our solution demonstrates robust and driftless real-time camera tracking in a television production studio environment.

Keywords: Dense Tracking, Dense 3D Reconstruction, Augmented Reality, Real-Time, RGB-D, GPU

1. Introduction

In this work, we develop a real-time image-based camera tracking solution for television broadcasting studios. Such a system enables adding virtual scene elements and interactive characters into live television broadcasts. The film industry knows this technique as *matchmoving*, which is traditionally done in post-processing using semi-automatic trackers [1]. A tracker is a tool which estimates a 3D camera motion trajectory based on observed 2D point tracks [2]. A tracker should only use the static scene geometry in order to avoid estimation bias. Scene segmentation is not typically fully automatic process and studio-specific standard technologies such as bluescreens, image masks and movement zones are used. Online matchmoving solutions exist, such as MotionAnalysis motion capture system, which tracks a camera in real-time based on passive markers attached to the camera. Brainstorm Multimedia product is compatible with many camera trackers and can be used to render view-dependent graphics in real-time. Although motion capture is precise and enables a large operating volume, the total price of such a system is currently 200 – 500k€. Overall, matchmoving is a time-consuming process, which is expensive and lacks automatic, affordable, and easy-to-use tools.

The main aim in this paper is to reduce the costs of matchmoving in studio environments by introducing an affordable RGB-D sensor based camera tracking tool which operates in real-time, fits studio use, and merely requires a low-end GPU. Our contributions are the following. 1) A RGB-D keyframe-based tracking method is proposed, which does not suffer from time-evolving global drift. A static studio scene is first modeled as a database of RGB-D keyframes, which are obtained by using our dense tracking approach and bundle adjustment. The database is then used as a reference for real-time pose estimation (Figure 1)[1]. By defining the camera tracking problem relative to the nearest keyframe, the time-evolving drift present in various other systems is avoided. We show how the keyframe tracking eventually outperforms incremental dense tracking. The estimation is robust due to gradient-based pixel selection and a M-estimator.

When aiming at processing dense RGB-D data in real-time, the algorithms must be parallelized to obtain sufficient performance and scalability properties. 2) The

[1] http://youtu.be/L_OLnFc7QxU

Figure 1: Top: A RGB sensor is used to build a keyframe database which contains the views in the camera motion zone. The TV camera pose is estimated by registering the image content with the nearest keyframe. Bottom: A real-time AR broadcast with virtual 3D characters which are rendered from the estimated pose. See example video[1].

computational scalability and performance of the camera tracking is improved by designing the full algorithm for a commodity GPU. A detailed description from our cost function definition into an efficient GPU implementation is given. 3) With a static keyframe-based 3D model available, the dynamic foreground points can be rejected from the camera pose estimation by observing discrepancies in intensity and depth simultaneously. Generally in RGB-D tracking, outliers both in color and depth can exist (occlusions, foreground objects, lighting effects etc) and must be taken care of. Our results are verified in a real television broadcasting studio with and without foreground dynamics. The proposed tracking solution is also compared with KinectFusion which is a recently presented alternative technique [3]. This work optimizes the previous work with near real-time simulation results [4] into real-time application use. Due to keyframes, the local drift will be negligible without depth map registration.

2. Related work

In this section, various image content based camera tracking approaches are discussed. Our focus is on affordable, real-time methods which can be robust in studio environments without extensive scene manipulation.

2.1. *Sparse feature based SLAM*

Visual Simultaneous Localization and Mapping (vSLAM) methods aim at estimating camera poses and structure concurrently from a real-time image sequence. The same scene geometry may be visible only for short time when the camera is moving. Therefore, an optimal pose and structure configuration are commonly solved within a sliding window of n recent frames using bundle adjustment [2, 5]. The problem is formulated by a cost function which measures 2D projection discrepancy of the current pose and the structure configuration. The minimization requires an initial guess, which is iteratively improved until the cost decreases below a threshold. The relative pose between two views are often inferred from *the essential matrix*, which can be estimated from 2D point correspondencies. The structure can be initialized using direct depth measurements or by triangulation from 2D projection points. This pose estimation process suffers from some drawbacks. Feature point extraction and matching is an error-prone process and requires an outlier rejection mechanism such as RANSAC. Extraction of high quality features such as SIFT can be expensive for real-time systems and still mismatches can not be fully prevented. For example, repetitive patterns and homogeneous textureless regions easily produce mismatches. Typically the estimation suffers from time evolving drift, which is often corrected by a loop closure mechanism. In loop closure, the current viewpoint is identified using a previously stored map. The cumulated pose error can then be divided evenly along the loop for removing the drift. Loop closure mechanisms are problematic in real-time AR applications unless they remove drift nearly in every frame, because bigger corrections are often visually disturbing. PTAM system by Klein is able to do loop closure for each frame, but suffers from error-prone feature extraction and matching process [6].

2.2. *Pose tracking using planar surfaces*

By introducing *a priori* information of surrounding 3D geometry into the estimation, the camera tracking becomes easier to solve. Several algorithms have been developed to track a camera based on a set of known 3D planar surfaces [7]. The planarity of the environment allows describing the optical flow analytically via

homography mapping, which can be decomposed into three-dimensional rotation and translation components. Homography mapping is described as a 3×3 matrix $\mathcal{H}$, which is traditionally solved from point correspondencies. After successful initialization, the pose increments can also be obtained by using homography-based warping function which minimizes appearance based cost function. Accurate camera tracking is possible using visible planar markers, but unfortunately they are visually disturbing during television broadcasting.

2.3. Direct methods with dense 3D structures

The direct methods use image alignment methods for pose estimation instead of depending on 2D feature points [4, 8, 9]. By avoiding the error-prone feature extraction and matching process, precision and robustness can be increased. Direct methods minimize the photometrical error between a reference point cloud and the current RGB image. The drift can be further reduced by matching also the current depth map values with the reference points [4]. A joint SLAM cost function has been formulated, which integrates structure component into the same cost function [10]. Steinbrücker *et al.* have recently proposed minimization of a photometrical cost [11], but do not take the benefit of M-estimator, which can increase robustness with neglible computational cost. The image warping function is used to model the appearance change due to camera motion. The warping requires a homography or a relatively accurate 3D estimate of the surrounding environment geometry. 3D point appearance generally depends on the lighting conditions and the point of view, but when observing subsequent frames, the points approximately maintain the same color. DTAM applies the photometrical cost function for pose estimation, but also refines 3D structure by minimizing photometrical reprojection error [12]. DTAM is designed to operate with monocular camera in real-time, but it has several drawbacks, such as a special initialization procedure, discretization of depth values, not being compatible with dynamic scenes, heavy hardware requirements and parameter tuning with the correct surface smoothness. A RGB-D sensor such as Microsoft Kinect is better suited for direct pose estimation because it produces a real-time feed of RGB-D measurements which can be photometrically aligned [13]. KinectFusion formulates pose estimation as depth map alignment problem using traditional Iterative Closest Point (ICP) [3]. The system also aims at maintaining small drift by registering the current depth maps with a voxel grid based 3D model, which is filtered over time by weighted averaging.

3. Dense RGB-D tracking

Our previous work tracked camera with respect to the RGB-D measurements which were recently captured by a RGB-D sensor. In this work, the previous frame reference is replaced by fixed keyframes which are stored in advance to avoid drift. The pose estimation is defined as a direct image registration task between the current image $\mathcal{I} : \mathbb{R}^2 \Rightarrow \mathbb{R}$ and the nearest keyframe $\mathcal{K} = \{\mathcal{P}^*, \mathbf{c}^*, \mathbf{T}^*\}$, where $\mathcal{P}^* = \{\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n\}$ is a set of 3D points and $\mathbf{c}^* = \{c_1, c_2, \ldots, c_n\}$ are the corresponding color intensities. The 4×4 matrix $\mathbf{T}^*$ defines the keyframe pose relative to the reference coordinate system. $\mathbf{T}^*$ maps points from the reference to the keyframe coordinate system. $*$ is used to denote fixed reference variables.

Our goal is to find the correct pose increment $\mathbf{T}(\mathbf{x})$ which minimizes scalar cost function $\frac{1}{2}\mathbf{e}(\mathbf{x})^T\mathbf{e}(\mathbf{x})$, where the residual is defined by

$$\mathbf{e} = \mathcal{I}\left(w\left(\mathcal{P}^*; \mathbf{T}(\mathbf{x})\widehat{\mathbf{T}}\right)\right) - \mathbf{c}^*. \tag{1}$$

$\widehat{\mathbf{T}} = \mathbf{T}^g(\mathbf{T}^*)^{-1}$ is the base transform, which encodes the relative pose between the initial guess $\mathbf{T}^g$ and the keyframe pose $\mathbf{T}^*$. $w(\mathcal{P}; \mathbf{T})$ is the warping function which transforms and projects 3D points by

$$w(\mathcal{P}; \mathbf{T}) = w(\mathcal{P}; \begin{bmatrix}\mathbf{R} & \mathbf{t} \\ 0 & 1\end{bmatrix}) = \mathbf{K}D(N(\mathbf{R}\mathcal{P} + \mathbf{t}), \boldsymbol{\alpha}), \tag{2}$$

where $N(\mathbf{p}) = (p_1/p_3, p_2/p_3)$ dehomogenizes a point, $D(\mathbf{p}, \boldsymbol{\alpha})$ performs lens distortion using parameters $\boldsymbol{\alpha} \in \mathbb{R}^5$ (constant) and $\mathbf{K}$ is a 3×3 intrinsic matrix of the camera (constant). We use standard Caltech distortion model

$$D(\mathbf{p}, \boldsymbol{\alpha}) = \begin{bmatrix}\mathbf{p}(1 + \alpha_1 r^2 + \alpha_2 r^4 + \alpha_5 r^6) + \mathbf{dx} \\ 1\end{bmatrix} \tag{3}$$

where $r^2 = p_1^2 + p_2^2$ and $\mathbf{dx}$ is the tangential distortion

$$\mathbf{dx} = \begin{bmatrix}2\alpha_3 p_1 p_2 + \alpha_4(r^2 + 2p_1^2) \\ \alpha_3(r^2 + 2p_2^2) + 2\alpha_4 p_1 p_2\end{bmatrix}. \tag{4}$$

$\mathbf{T}(\mathbf{x}) \in \mathbb{SE}(3)$ forms a Lie group by exponential mapping [14]:

$$\mathbf{T}(\mathbf{x}) = e^{\mathbf{A}(\mathbf{x})}, \quad \mathbf{A}(\mathbf{x}) = \begin{bmatrix}[\boldsymbol{\omega}]_\times & \boldsymbol{\upsilon} \\ \mathbf{0} & 0\end{bmatrix}, \tag{5}$$

where $\mathbf{x} = (\boldsymbol{\omega}, \boldsymbol{\upsilon}) \in \mathbb{R}^6$ encodes relative 3D motion between two camera poses. $\boldsymbol{\omega}$ is the rotation axis and $\|\boldsymbol{\omega}\|$ is the rotation angle in radians, and $\boldsymbol{\upsilon}$ is the velocity twist. $[\,]_\times$ produces a skew-symmetric matrix which

performs a cross product. The matrix exponential produces 3×3 rotation matrix $\mathbf{R}$ and 3×1 translation vector $\mathbf{t}$ embedded in a 4×4 matrix. This cost function form does not require any predefined temporal correspondencies. The warping function will produce the dense correspondency when the minimization has converged.

3.1. Minimization

The inverse compositional approach [9] is adopted for efficient minimization of the cost function. The cost is reformulated as

$$\begin{aligned}\mathbf{c}^*(\mathbf{x}) &= \mathcal{I}^*\left(w\left(\mathcal{P}^*; e^{-\mathbf{A}(\mathbf{x})}\right)\right),\ \mathbf{c}^w = \mathcal{I}\left(w\left(\mathcal{P}^*; \widehat{\mathbf{T}}\right)\right)\\ \mathbf{e} &= \mathbf{c}^*(\mathbf{x}) - \mathbf{c}^w,\end{aligned}$$

where reference colors in $\mathbf{c}^*$ are now a function of the inverse motion increment (Figure 2). The current warped intensities $\mathbf{c}^w$ are produced by resampling $\mathcal{I}$ using $\widehat{\mathbf{T}}$. The residual $\mathbf{e}$ is then minimized by resampling the reference image $\mathcal{I}^*$. The benefit can be observed from the form of the Jacobian

$$\mathbf{J}_{ij} = \frac{\partial \mathbf{c}(\mathbf{0})}{\partial \mathbf{x}} = \nabla\mathcal{I}^*\left(w(\mathbf{P}_i;\mathbf{I})\right)\frac{\partial w(\mathbf{P}_i;\mathbf{I})}{\partial x_j}, \qquad (6)$$

where $\nabla\mathcal{I}(\mathbf{p}) = [\ \frac{\partial\mathcal{I}(\mathbf{p})}{\partial x}\ \ \frac{\partial\mathcal{I}(\mathbf{p})}{\partial y}\ ]$. Now $\mathbf{J}$ does not depend on $\widehat{\mathbf{T}}$ anymore and it can be computed only once for each $\mathcal{K}^*$ for better performance. Gauss-Newton iteration is well-suited for minimization when the initial guess is near the minimum and the cost function is smooth. The cost function is locally approximated by the first-order Taylor expansion $\mathbf{e}(\mathbf{x}) = \mathbf{e}(\mathbf{0}) + \mathbf{J}\mathbf{x}$, where $\mathbf{e}(\mathbf{0})$ is the current residual. $c_k^*(\mathbf{0})$ are bi-linearly interpolated only once using reference image $\mathcal{I}^*$. The scalar error function becomes

$$\frac{1}{2}\mathbf{e}(\mathbf{x})^T\mathbf{e}(\mathbf{x}) = \frac{1}{2}\mathbf{e}(\mathbf{0})^T\mathbf{e}(\mathbf{0}) + \mathbf{x}^T\mathbf{J}^T\mathbf{e}(\mathbf{0}) + \frac{1}{2}\mathbf{x}^T\mathbf{J}^T\mathbf{J}\mathbf{x} \quad (7)$$

where the derivative is zero when

$$\mathbf{J}^T\mathbf{J}\mathbf{x} = -\mathbf{J}^T\mathbf{e}(\mathbf{0})\ . \qquad (8)$$

Because $\mathbf{J}^T\mathbf{J}$ is a 6×6 positive definite matrix, the inversion can be efficiently calculated using the Cholesky method. The matrix multiplication associativity enables collecting the estimated increment into the base transform by $\widehat{\mathbf{T}}^k e^{\mathbf{A}(\widehat{\mathbf{x}})} \Rightarrow \widehat{\mathbf{T}}^{k+1}$.

3.2. Cost function smoothness

The first-order Taylor approximation is required for efficient minimization and therefore the cost function must be assumed to be locally smooth. Unfortunately,

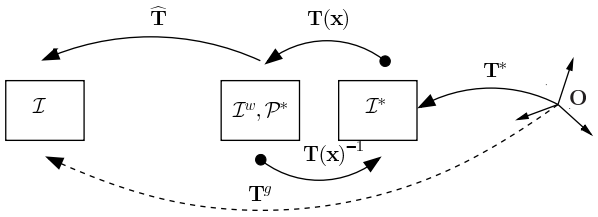

Figure 2: Inverse compositional image alignment. The inverse increment is found which minimizes the difference between $\mathcal{I}^w$ and $\mathcal{I}^*$. Linearization is specially computed only once using $\mathcal{I}^*$ and $\mathcal{P}^*$.

the exact cost function is non-linear and non-continuous due to arbitrary surface and appearance functions and the perspective projection. Bi-linear interpolation is used for continuous sampling of the images and smoothness is assumed between the pixels. The motion is parameterized by the matrix exponential, whose derivatives at zero are simply constant matrices $\frac{\partial\mathbf{T}(\mathbf{0})}{\partial x_j} = \frac{\partial\mathbf{A}(\mathbf{0})}{\partial x_j}$. Strong photometrical variations at different resolutions are the main cue of camera motion. The cost smoothness increases with the number of 3D points because noise deviations average out. Larger convergence domains can be obtained by using downsampled images, provided that aliasing effects do not dominate. By experimentation, 80×60 images provide rough estimates which can be refined using higher resolution images. Discontinuities in the cost function may occur when switching between different resolutions. Maintaining a fixed structure throughout the minimization is important and thus we do not downsample the geometry even though low resolution intensity images are used.

3.3. Rejecting outliers using M-estimator

The warping function models the appearance change due to camera motion when 3D points are static with Lambertian reflectance. Some of the points are considered outliers, because this model is not sufficient to explain their appearance. For example, intensity changes due to occlusions, lighting effects and geometry dynamics have not been modeled. Outliers can be detected efficiently by comparing the intensity and depth values at the warped points and the reference points. Inliers always have small error whereas outliers may have any error value. Increased robustness is obtained by damping the high error values out from the estimation (Figure 3). Instead of using a fixed threshold, a certain adaptation level to image brighness variations is useful. One method to determine the damping weights is by the

Tukey weighting function

$$u_k = \frac{|e_k|}{c * median(\mathbf{e}_a)}, \quad (9)$$

$$\mathrm{w_k}^c = \begin{cases} (1-(\frac{u_k}{b})^2)^2 & \text{if } |u_k| <= b \\ 0 & \text{if } |u_k| > b \end{cases}, \quad (10)$$

where $\mathbf{e}_a = (|e_1|, |e_2|, \ldots, |e_n|)$ is a vector of absolute residual values, $c = 1.4826$ is the robust standard deviation, and $b = 4.6851$ is the Tukey specific constant. Thus, Tukey weighting generates adaptive weighting based on the statistical distribution of the residual.

The depth correlation weights are computed directly from the depth residual

$$\mathbf{e}_z = \mathcal{Z}\left(w(\mathcal{P}; \widehat{\mathbf{T}})\right) - \mathbf{e}_3^T \widehat{\mathbf{T}} \begin{bmatrix} \mathcal{P} \\ 1 \end{bmatrix}, \quad (11)$$

without adaptation, where $\mathcal{Z} : \mathbb{R}^2 \Rightarrow \mathbb{R}$ is the depth map function of the current RGB image and $\mathbf{e}_3^T = (0, 0, 1, 0)$ is used to obtain the current depth value. When the standard deviation of depth measurements is τ, the warped points whose depth differs more than τ from the current depth map value can be interpreted as foreground objects. The depth weights are determined by

$$\mathrm{w_k}^z = max\left(1 - \mathbf{e}_z^2(k)/\tau^2, 0\right)^2. \quad (12)$$

The weighted Gauss-Newton step is obtained by rewriting equation 8 as follows

$$\mathbf{J}^T\mathbf{W}\mathbf{J}\mathbf{x} = -\mathbf{J}^T\mathbf{W}\mathbf{e}, \quad (13)$$

where $\mathbf{W}$ is a diagonal matrix with $\mathrm{diag}(\mathbf{W})_k = \mathrm{w_k}^c \mathrm{w_k}^z$. The robust step is obtained by $\mathbf{J} \Leftarrow \sqrt{\mathbf{W}}\mathbf{J}$ and $\mathbf{e} \Leftarrow \sqrt{\mathbf{W}}\mathbf{e}$. The weights are quadratic and therefore square root is not evaluated in practice.

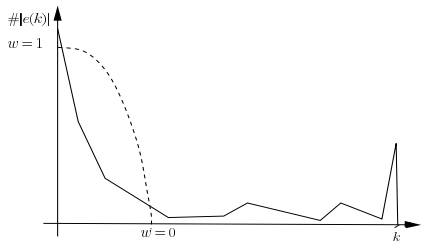

Figure 3: Tukey M-estimator effect illustrated. The points with high error values are given small or zero weight, because their appearance change is not explained by the motion model.

3.4. Selecting reference points

The points used in the motion estimation can be freely selected from a reference RGB-D measurement. The points which do not contribute to the residual vector through linearization are not useful. Thus, only the points, which are associated with the greatest absolute values of the Jacobian should be selected [15]. The selected points will then have strong optical flow $\|\frac{\partial w}{\partial \mathbf{x}}\| > \theta_w$ and/or strong image gradient $\|\frac{\partial \mathcal{I}}{\partial \mathbf{p}}\| > \theta_g$. We use a simpler selection criteria and focus on the set of points $\mathcal{P}_s^*$ which merely have strong image gradient. This method fits GPU better, because it does not require sorting of the Jacobian elements.

$$\mathcal{P}_s^* = \{\, \mathbf{P}_k \mid \|\nabla \mathcal{I}^*(w(\mathbf{P}_k; \mathbf{I}))\| > \theta_g \,,\, \mathbf{P}_k \in \mathcal{P}^* \}, \quad (14)$$

where θ_g is the threshold which selects p percent of all points. p depends on the computational capacity available and the amount of image edges in the application context.

3.5. Point distortion modeling vs image undistortion

By modeling the distortion in the warping function, the input images do not need to be undistorted. This is beneficial because image undistortion requires additional resampling of the image. Resampling degrades raw RGB data, because bi-linear interpolation fails to predict exact colors precisely. Also the number of distortion operations for a set of points will be smaller than full image resolution even though warping has to be done many times per frame.

4. Real-time implementation

Our method was implemented in Ubuntu Linux environment using open software tools, Microsoft Kinect and a commodity PC laptop hardware on which the method runs real-time. The cost minimization requires special attention when aiming at an efficient real-time implementation. Because the computational phases benefit from parallel computing, we implemented the full method (algorithm 1) on a commodity GPU. The same minimization algorithm can be used with dense incremental tracking and keyframe tracking. The implementation scales into multiple threads/cores. Only Cholesky inversion and matrix exponential computation are executed on a single GPU thread.

Algorithm 1 Minimization on GPU.

Input: $\mathcal{I}_{L=\{1,2,3\}}$ with 320×240, 160×120, and 80×60 sizes. $\mathcal{Z}$ in 320×240. Iteration counts $\{n_1, n_2, n_3\}$. $\widehat{\mathbf{T}} = \mathbf{T}_0$.
Output: Relative pose $\widehat{\mathbf{T}}$.

for all multi-resolution layers $L = \{3, 2, 1\}$ **do**
  for all iterations $j = \{1 \dots n_L\}$ **do**
    Compute residual $\mathbf{e}$ and $\mathbf{W}_z$ (sec. 4.1)
    Determine M-estimator weights $\mathbf{W}_c$ (sec. 4.2)
    $\mathbf{W} \Leftarrow \mathbf{W}_c * \mathbf{W}_z$
    $\mathbf{J} \Leftarrow \sqrt{\mathbf{W}}\mathbf{J}$, $\mathbf{e} \Leftarrow \sqrt{\mathbf{W}}\mathbf{e}$
    Reduce linear system (sec. 4.3)
    Solving linear system for $\widehat{\mathbf{x}}$ (sec. 4.3)
    $\widehat{\mathbf{T}} \Leftarrow \widehat{\mathbf{T}} e^{\mathbf{A}(\widehat{\mathbf{x}})}$ (sec. 4.4)
  end for
end for

4.1. Warping

The warping function is applied in parallel to the selected $\mathbf{P}_k \in \mathcal{P}^*$. The cost function (eq. 1) is evaluated using bilinear interpolation. Additionally the points are transformed into IR view for evaluating the nearest depth value (eq. 11). Unnecessary resampling errors are avoided by evaluating the depth values directly in the IR view. The depths are used to generate weighting $\mathbf{W}_z$. The warping is illustrated in Figure 4.

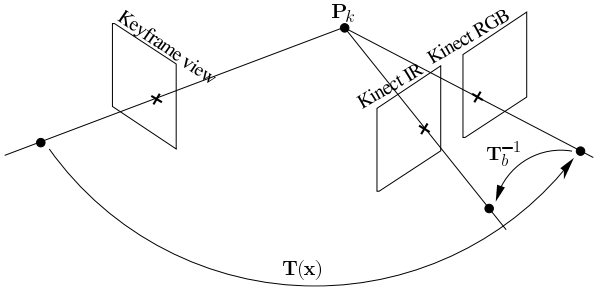

Figure 4: The warping of points from keyframe into Microsoft Kinect RGB and IR image coordinate systems. The additional depth lookup from an IR image is required by Equations 1 and 11.

4.2. M-estimator

Tukey based weights require the median of the error distribution, which easily becomes a bottleneck of using M-estimator. Therefore, we use the histogram technique (eq. 15) to find an approximate median. The distribution of the residual is represented using a 64-bin histogram and n is set to half residual length. `histogram64` routine fits this purpose, because it is fast enough to be executed in every iteration and is specifically designed for NVIDIA video cards [16]. By experiment, 64 bins seems to provide sufficient adaptation to different error distribution profiles.

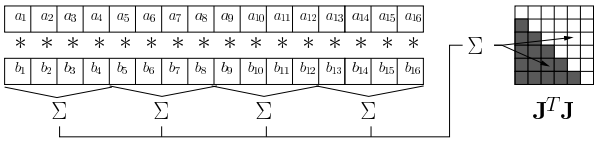

Figure 5: Efficient reduction on GPU. Dot products are divided into sub-ranges which are reduced in parallel. $\mathbf{J}^T\mathbf{J}$ is symmetric 6×6 matrix and elements can be mirrored with respect to the diagonal.

4.3. Linear system reduction

The reduction of the linear system means computing $\mathbf{J}^T\mathbf{J}$ and $\mathbf{J}^T\mathbf{e}$. This phase compresses the linear equations from n-dimensional to six-dimensional. Reduction is required for each iteration and therefore should be implemented efficiently. Matrix multiplication is often computed in parallel by dividing k dot products into separate threads. In this case, we have only 36 dot products, but our video card can manage 1024 threads in parallel. To gain maximal efficiency, we parallelize the computation in the dot product direction instead. Each dot product has n elements, where n is usually 8192 or more. Dot products are reduced by dividing them into sub-blocks and summing them efficiently in parallel. The total sum is finally cumulated from the subsums. $\mathbf{J}^T\mathbf{J}$ is a positive semidefinite matrix which is also symmetric. This means it is sufficient to compute only the upper triangle of the values and mirror the results into the lower triangle. This property reduces the number of dot products from 36 to 21. $\mathbf{J}^T\mathbf{e}$ requires six dot products. The process is illustrated in Figure 5. The inversion can be efficiently calculated using Cholesky decomposition due to positive definite property. The explicit inverse of $\mathbf{J}^T\mathbf{J}$ can also be avoided by using the conjugate gradient method with six iterations. Both approaches are fast due to the small matrix size. The inversion is executed on a single GPU thread.

4.4. Evaluating matrix exponential

Matrix exponentials can be computed in closed form using the translation extended *Rodriquez formula* [14]. Unfortunately it is not numerically stable in the most important small angle case. Expokit, however, provides various numerical approaches which are guaranteed to produce smooth and precise mapping [17]. By comparing them with MATLAB default implementation, it seems that complex matrix exponential (`zgpadm`) is the most accurate. Thus, we choose to evaluate $\mathbf{T}(\mathbf{x}) = e^{\mathbf{A}(\mathbf{x})}$

by the matrix exponential of a general complex matrix in full, using the irreducible rational Padé approximation combined with scaling-and-squaring. The exponential is computed in a single GPU thread due to sequential nature of the operation.

4.5. Selecting points on GPU

Even though keyframe reference points can be selected in a pre-process, a fast implementation is useful when executing dense tracking incrementally. We adopt the histogram technique to find the best points efficiently [4]. We seek such histogram bin B_t for which

$$\Sigma_{k=B_t-1}^{255} \mathbf{h}(k) < n <= \Sigma_{k=B_t}^{255} \mathbf{h}(k), \qquad (15)$$

where n is the number of points to be selected and

$$\begin{aligned} \mathcal{G} &= |\nabla_u \mathcal{I}^* (w(\mathcal{P}^*; \mathbf{I}))| + |\nabla_v \mathcal{I}^* (w(\mathcal{P}^*; \mathbf{I}))| \\ \mathbf{h} &= \text{histogram}(\mathcal{G}/2.0) \end{aligned}$$

The value range is bounded to $[0, 255]$ which fits `histogram256` method designed for NVIDIA video cards [16]. The final n indices are collected into a packed array. Packing is important because it allows eliminating all non-interesting points from further processing in the pipeline. Packing is implemented on a GPU by assigning each thread a slice of the original index range to compress (Figure 6). Each thread stores the interesting points into a temporary packed buffer with the count. The counts must be collected from all threads to determine the final output range of each thread. This allows parallel point selection which scales into multiple threads. Due to the discretization into bins, n pixels may not match the bin boundary automatically. n should be a multiple of 1024 to match the maximum amount of threads on a GPU. This is why it is useful to classify points into

$$\begin{aligned} \mathcal{P}_{semi} &= \{ |\nabla c_k| = B_t \mid \mathbf{P}_k \in \mathcal{P}^* \} \\ \mathcal{P}_{good} &= \{ |\nabla c_k| > B_t \mid \mathbf{P}_k \in \mathcal{P}^* \}, \end{aligned}$$

where $\mathcal{P}_{good}$ are all selected and the remaining points are chosen from $\mathcal{P}_{semi}$ to obtain exactly n selected points.

4.6. Preprocessing RGB images

The preprocessor converts 640×480 Microsoft Kinect Bayer images into a pyramid of 320×240, 160×120, and 80×60. A 5×5 Gaussian filter is used with downsampling of the high resolution images. Downsampling is almost lossless, because the Bayer images are redundant. 2×2 block averaging is used to produce the rest of the layers. The lower resolution x and y coordinates are obtained by $x_L = \frac{1}{2^L} x + \frac{1}{2^{L+1}} - \frac{1}{2}$, where L is the amount of layers in-between. The RGB pre-processing steps are per-pixel operations which are executed in separate threads on a GPU.

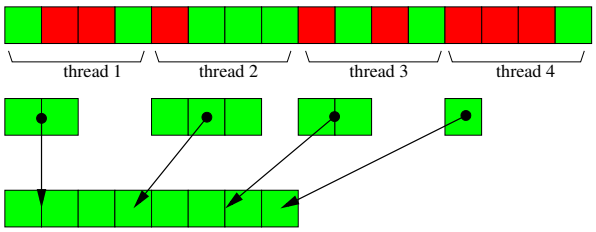

Figure 6: Index buffer packing illustrated. Each thread selects points from a sub-range. The sub-counts are shared between the threads and the output buffer written.

4.7. Microsoft Kinect Calibration

Microsoft Kinect is conveniently calibrated using a specialized toolbox by Herrera *et al.*, which jointly estimates all calibration parameters [18]. A chessboard pattern is printed and a set of matching raw disparity images and RGB images are captured and loaded into the toolbox. The toolbox then semi-automatically provides the intrinsic matrices $\mathbf{K}_{IR}$, $\mathbf{K}_{RGB}$, the baseline transform $\mathbf{T}_b$ between the views, reconstruction parameters $(c_0, c_1) \in \mathbb{R}^2$ and distortion coefficients $\boldsymbol{\alpha}_{RGB}$, α_0, α_1 and $\boldsymbol{\beta}$. The raw disparity map $\mathcal{D}(u, v)$ is then undistorted by

$$u(d, \alpha_0, \alpha_1, \boldsymbol{\beta}) = d + \boldsymbol{\beta}(u, v) \exp(\alpha_0 - \alpha_1 d), \qquad (16)$$

and converted into depth values by

$$z(d) = \frac{1}{c_0 + c_1 u(d)}. \qquad (17)$$

The disparity undistortion model corrects exponential decay of accuracy, but also models per pixel distortions using a map $\boldsymbol{\beta}(u, v)$. The RGB image lens distortion is modeled using the standard Caltech parameters $\boldsymbol{\alpha}_{RGB} \in \mathbb{R}^5$, which models radial distortions using three coefficients and tangential distortions using two coefficients [19]. Typically only the first two radial components are necessary, and the rest can be fixed to zero to speed up computation. The toolbox allows fixing distortion parameters prior to calibration.

4.8. Generating point cloud from raw disparity map

The 3D points are generated in parallel by a baseline transform

$$\mathbf{P}_k = \mathbf{T}_b z(d_k) \mathbf{K}_{IR}^{-1} (\mathbf{p}_k \; 1)^T, \qquad (18)$$

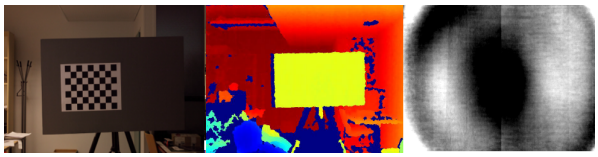

Figure 7: RGB and disparity images, and distortion pattern β [18]

where $\mathbf{p}_k = (u_k, v_k)$ are the pixel coordinates in the IR view, $\mathbf{K}$ is the intrinsic matrix of the IR view, and the 4×4 baseline matrix $\mathbf{T}_b$ maps points from the IR view into the RGB view. Each cloud $\mathcal{P}$ has storage for 320×240 points which are processed in parallel and stored in a linear array. Further compressing of point clouds is done through point selection procedure (sec. 4.5). The intensity vector $\mathbf{c}^*$ matching with the 3D points $\mathbf{P}_k \in \mathcal{P}$ is produced by bi-linear interpolation, because the points do not match with RGB image pixels.

4.9. Results

The minimization algorithm is implemented on a low-end NVIDIA NVS4200m GPU using CUDA. The GPU has 48 CUDA cores and it allows executing 1024 parallel threads on a single multi-streaming processor. Because our implementation divides the computational task into n threads, it is scalable and benefits from GPU hardware development. Currently the most powerful video cards, such as the NVIDIA Tesla K10, have 3072 CUDA cores. The computation time of the processing phases is illustrated in Figure 8. The green bars represent the minimization phases which directly scale into n threads and therefore become faster with more powerful GPU. 8192 points are selected and the minimization uses three multi-resolution layers 80×60, 160×120 and 320×240. The corresponding iteration counts are 2, 3, and 10. In effect, the system operates at 30Hz and the computation takes 26ms which leaves 7ms for rendering the augmented graphics.

5. Incremental Dense Tracking Experiments

The incremental dense tracking is sketched in Algorithm 2. In this section the dense tracking accuracy is evaluated using the RGB-D SLAM benchmark provided by Technical University of Münich [20]. Kinfu is the open source implementation of KinectFusion [21] which is a recent technique for low-cost camera tracking and reconstruction. The proposed dense RGB-D tracking accuracy is compared with Kinfu (Figure 9). Kinfu experiments were executed on a powerful workstation GPU (NVIDIA Quadro 2000, 1024MB, 192 CUDA cores) almost in real-time whereas our approach was executed in real-time on a low-end laptop GPU (NVIDIA NVS4200m, 1024MB, 48 CUDA cores). At this point, keyframes are not used and time-evolving drift is present in both systems. Incremental dense tracking uses recent RGB-D measurements as motion reference whereas Kinfu incrementally reconstructs a voxel-based 3D model of the scene which is used as motion reference. Kinfu uses ICP to estimate camera pose, which is based on a fully geometrical distance function.

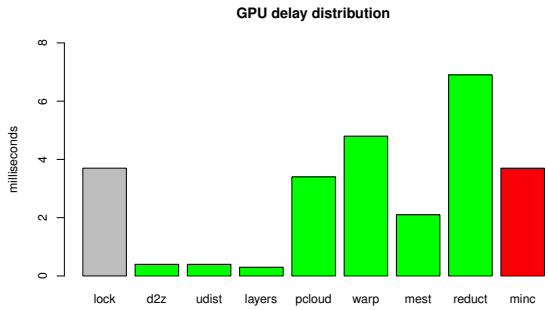

Figure 8: GPU computation time for one frame. Green bars represent phases which are scalable to n threads. The red bar must be executed on a single thread. The gray bar represents the CUDA interop delay of buffer locking which is required for visualization only. `d2z` is the disparity map to zmap conversion, `layers` is the conversion to grayscale multi-resolution format, `pcloud` is the 3D reconstruction, `warp` contains the cost function evaluations, `mest` is the M-estimation, `reduct` is the reduction of the linear system, and `minc` contains the delay of solving the motion parameters and evaluating the matrix exponential.

Algorithm 2 Incremental dense tracking algorithm

Require: $\{\mathcal{P}^*, \mathbf{c}^*\} \Leftarrow$Select the best points from 1st RGB-D.
Input: $\mathbf{T}_{cur} = \mathbf{T}_{ref} = \mathbf{I}$
Output: Trajectory $\{\mathbf{I}, \mathbf{T}^1_{cur}, \ldots, \mathbf{T}^n_{cur}\}$.

for each RGB-D measurement **do**
  $\widehat{\mathbf{T}} \Leftarrow$ Minimize$(\mathbf{I}, \mathcal{P}^*, \mathbf{c}^*, \mathcal{I}_{cur})$
  $\mathbf{T}_{cur} \Leftarrow \widehat{\mathbf{T}}\mathbf{T}_{ref}$
  if reference update signaled **then**
    $\{\mathcal{P}^*, \mathbf{c}^*\} \Leftarrow$ Select the best points
    Precompute Jacobian
    $\mathbf{T}_{ref} = \mathbf{T}_{cur}$
  end if
end for

Table 5 shows the comparison between our method and Kinfu numerically using two Freiburg sequences with known ground truth trajectory. Our dense tracking drifts $1.08cm/s$ with the slower `freiburg2/desk` sequence and $2.60cm/s$ with the faster `freiburg1/desk` sequence. Kinfu has smaller drift with small voxel volumes (such as $(3m)^3$) when the scene geometry is versatile, but seems to suffer from gross tracking failures

with bigger volumes such as $(5m)^3$ and $(8m)^3$. Operation volume is limited because 512^3 voxel grid becomes too coarse and bigger grids not not fit into GPU memory. Also ICP breaks down easily when the scene contains mostly planar surface (e.g. floor) [1]. Drift was measured by dividing the input frames into subsegments of several seconds (10 and 2 correspondingly) whose median error was measured against the ground truth. One second average error was computed from the subsegment with median error to average out random perturbations and to neglect Kinfu tracking failures, which occurred in all cases except on `freiburg2/desk` using $(3m)^3$ volume. Our incremental dense tracking has generally smaller drift with the faster `freiburg1/desk` sequence, but both Kinfu (with the most compatible grid) and our method lost track once during the sequence. In this case $(3m)^3$ grid does not contain versatile geometry and the result is worse than with bigger volumes. Re-localization issues are discussed later in the paper. Our method has more delay than previously because the reference frames must be updated more frequently with the faster sequence. Kinfu's dependency on careful setting of volume size, and dependency on geometrical variations makes it unsuitable to be used in our application. In television production studios, scenes can easily be larger than $(3m)^3$ and sufficient geometrical variation is more difficult to guarantee than sufficient texturing. Our dense tracking operates without failures even when planar surfaces are present, because our cost function matches also texturing. Memory consumption can be low even in larger operating volumes, because the keyframe placement can be optimized based on the camera motion zones.

Dataset	Our drift	Kinfu(3)	Kinfu(8)	Motion
`freiburg1/desk`	$2.60cm/s$	$8.40cm/s$	$3.97cm/s$	$41.3cm/s$
	$52.2ms$	$135ms$	$135ms$	
`freiburg2/desk`	$1.08cm/s$	$0.64cm/s$	$1.30cm/s$	$19.3cm/s$
	$35.5ms$	$135ms$	$135ms$	

Table 1: Drift and delay compared to Kinfu with $(3m)^3$ and $(8m)^3$ operating volume.

6. Dense RGB-D tracking using keyframes

Despite that our dense tracker works in cases where KinectFusion fails, the tracking suffers from time-evolving drift, which is not acceptable in studio use, because it causes virtual items to move away from their correct pose [4]. A fixed 3D model is required to avoid drift (Figure 1). A keyframe model can be generated with various offline/online techniques prior to broadcasting [2, 22]. The keyframe tracking is sketched in Algorithm 3. Tracking is initialized at the first keyframe whose $\widehat{\mathbf{T}} = \mathbf{I}$. The cost function (eq. 1) is then minimized in each frame to obtain 3D camera pose. The reference keyframe is switched when the current pose becomes closer to another keyframe. Keyframe tracking is driftless and very fast due to keyframe pre-computations and the utilization of GPU.

[1] Video: http://youtu.be/tNz1p1sdTrE

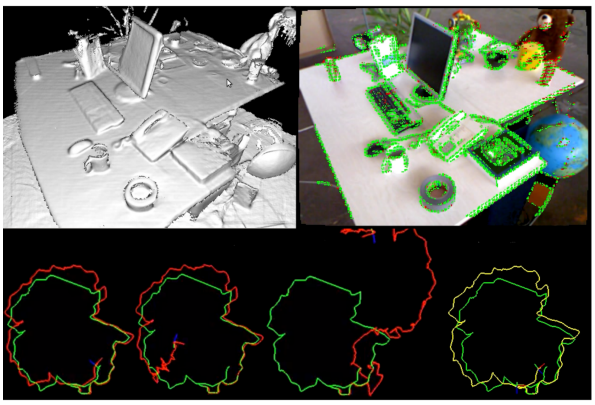

Figure 9: Kinfu performance compared with incremental dense tracking using `freiburg_desk2` sequence with motion capture ground truth (green trajectory). Kinfu is executed with $(3m)^3$, $(5m)^3$ and $(8m)^3$ voxel volumes. The red trajectories on the left are output from from Kinfu. Kinfu gains lower drift due to structure integration, but planar surfaces cause tracking failures. $(3m)^3$ volume does not contain floor and therefore Kinfu works well. On the right, the yellow trajectory is the proposed incremental RGB-D tracking result. Problems with planar surfaces do not exist, and the method allows larger operating volumes due to lower memory consumption. By using keyframes, the drift can be completely eliminated. The green dots reveal the selected points.

Algorithm 3 Keyframe tracking algorithm.

Require: Keyframe database available
Input: $\mathbf{T}_{cur} = \mathbf{I}$
Output: Trajectory $\{\mathbf{I}, \mathbf{T}^1_{cur}, \ldots, \mathbf{T}^n_{cur}\}$.

1: **for** each RGB-D measurement **do**
2: $\{\mathcal{P}^*, \mathbf{c}^*, \mathbf{T}^*\} \Leftarrow$FindKeyframe$(\mathbf{T}_{cur})$
3: $\widehat{\mathbf{T}} \Leftarrow$Minimize$(\mathbf{T}_{cur}(\mathbf{T}^*)^{-1}, \mathcal{P}^*, \mathbf{c}^*, \mathcal{I}_{cur})$
4: $\mathbf{T}_{cur} \Leftarrow \widehat{\mathbf{T}}\mathbf{T}^*$
5: **end for**

6.1. Keyframe model generation

In a studio environment, online structure correction mechanism is an unnecessary because the correct 3D model can be fixed prior to a broadcast. We are in-

terested in rapid and semi-automatic reconstruction approaches, which allow verifying and editing the 3D model prior to broadcasting. We sweep the scene using RGB-D sensor and concurrently store the pose parameters and depth maps using our real-time dense tracking approach (Algorithm 2). At this time, the studio scene must not contain moving actors but merely static geometry. The final keyframes are incrementally chosen from the RGB-D stream by discarding the frames which are too close to the existing database (Section 6.2). A loop-closing mechanism could be used with more complex sweeps where the camera re-enters same view points. Loop closure can be implemented by alternating between Algorithms 2 and 3 and adding a keyframe only when necessary. Both methods are rapid and they produce a sub-optimal set of keyframes, which can be further refined in offline when necessary.

A simple keyframe model generation process is

- Incremental dense tracking (short sweep)

Followed by the optional offline refinement phases

- Depth map fusion
- Sparse bundle adjustment (semi-automatic annotation of keypoints)

Because the keyframes are sparsely selected from the stream of RGB-D measurements, they do not utilize all information available. The intermediate point clouds, which are not selected as keyframes, are warped into a nearest keyframe and the final maps are filtered in post-processing. In effect, this improves depth map accuracy and fills holes. A large portion of the outlier points can be neglected based on color deviation to the keyframe pixels. The initial depth map is obtained as the median of the warped depths similar to Hirschmüller [23]. The inverse depth samples within $z_m \pm \delta$ are averaged to find the robust estimate (Figure 10). Inverse depths typically have Gaussian distribution in stereo based settings. δ is the depth sample window, which is user-specified. δ depends on the RGB-D sensor depth noise level. This procedure is comparable to the weighted sum of Truncated Signed Distance Functions [3], but the surface will always appear near the median distance, and therefore does not depend on the grid density.

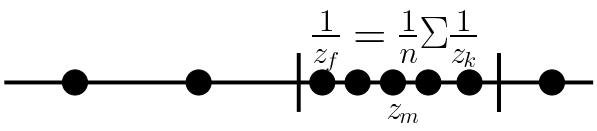

Figure 10: Robust depth estimation near median depth illustrated.

Sparse bundle adjustment can be used to remove the small global error at the end of estimated camera trajectory [24]. SBA does not converge without a good initial guess. Sometimes an initial guess can be succesfully extracted from an unordered image set by extracting and matching feature points and initializing camera configuration using their geometrical relations. The process is error-prone because image content such as homogeneous texturing and repetitive patterns prevents reliable extraction/matching. The studio scenes in our case do not contain much texture variations, and therefore automatic feature extraction fails. In this case, however, dense tracking provides very good initial guess of the pose configuration. To be able to execute SBA, few 3D points and their 2D correspondence are required from the user. An easy-to-use interactive editor was implemented for this purpose. Multiple geometries disappear when executing SBA (Figure 11).

Figure 11: The effect of sparse bundle adjustment illustrated. Minor error at the end of the trajectory is corrected in the right side image. The dense scene geometry becomes precise with correct poses. Notice especially the sofa shape before and after the correction.

6.2. *Finding nearest keyframe*

A similarity metric is required to find the nearest key pose $\mathbf{T}_k$. The challenge is to unify rotation and translation differences, because they are expressed in different units. First, we define the relative transformation

$$\Delta\mathbf{T}_k = \mathbf{T}_{cur}\mathbf{T}_k^{-1} \Rightarrow (\theta_k, \mathbf{v}_k, \mathbf{d}_k), \qquad (19)$$

where $\mathbf{T}_{cur}$ is the current camera pose. The relative rotation is decomposed into angle-axis representation $(\theta, \mathbf{v})$ based on the *Euler's rotation theorem*. The translation between the poses is expressed as the vector $\mathbf{d}$ between the origins. We define a potential keyframe index set as

$$\mathbf{\Omega} = \{\, |\theta_k| < \theta_{max}, \|\mathbf{d}_k\| < d_{max} \,\|\, k \in [1, n] \,\}, \qquad (20)$$

where n is the number of keyframes in the database. Thus $\mathbf{\Omega}$ contains a subset of keyframe indices whose angle and distance are below user defined thresholds θ_{max}

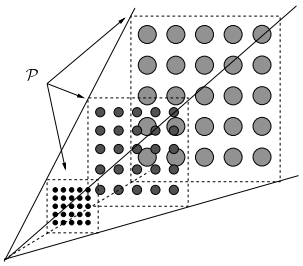

Figure 12: The 3D test point set $\mathcal{P}$ used to compare camera poses is illustrated. $\mathcal{P}$ approximates the view frustum by three 2D point grids. The projection error of $\mathcal{P}$ between the current view and each keyframe view is compared to find the nearest keyframe.

and d_{max}. This pre-selection prunes out distant poses efficiently. Thresholds are easy to set based on keyframe density in a 3D volume. The best keyframe is chosen by transforming the view frustum, represented by a set of 3D points, from the keyframe into the current frame and observing the 2D point discrepancy. This unifies rotation and translation errors into a single metric

$$s = \underset{k \in \mathbf{\Omega}}{\operatorname{argmin}} \|w(\mathcal{P}; \Delta\mathbf{T}_k) - w(\mathcal{P}; \mathbf{I})\|, \qquad (21)$$

where $\mathcal{K}_s$ is the nearest keyframe. In contrast to frustum intersection, this metric works also when the camera is rotating around z-axis. The test point set $\mathcal{P}$ is a sparse representation of the view frustum (Figure 12). In particular, the frustum is approximated by three sparse 2D grids, each having uniformly sampled depth coordinates in the overall depth range of the RGB-D sensor. The 3D points at different layers are generated by discrete steps along the viewing rays.

7. Dense keyframe tracking experiments

In Figure 13, it is shown how the drift increases with dense tracking when moving back and forth along a fixed rail in a studio environment. We measure distance to the ground truth in every frame as error metric. The ground truth is generated without bundle adjustment, simply by fitting a line based on the first sweep (alg. 2). Incremental tracking collects pose error from two different sources: 1) numerical inaccuracies in the pose estimation, 2) moving objects in the reference view. The drift problem is solved by tracking relative to keyframes which contain merely static background geometry (alg. 3). The demonstration video shows the difference between dense tracking with and without keyframes [3].

[3] http://youtu.be/zfKdZSkG4LU

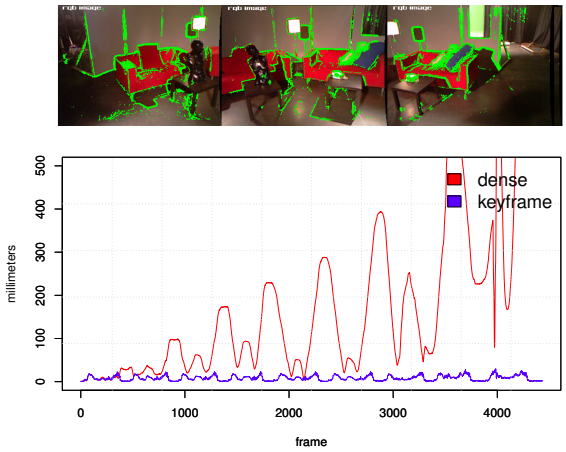

Figure 13: Camera moving forward and backward along a fixed 3.30m studio rail. The top figure shows three images taken from the beginning, middle and the end of the rail. Green region represents the selected points. On the bottom, the comparison between incremental tracking and keyframe tracking. In incremental tracking drift increases in time, but keyframe tracking maintains small bounded error. A person is moving in the scene during the last cycles.

Figure 14 shows how the online tracking accuracy depends on the number of keyframes. The sequence is illustrated in Figure 1, but we measure keyframe switching error in an empty scene for eliminating occlusion effects. The ground truth is generated by applying bundle adjustment to 27 keyframes which are initialized by dense tracking. Keyframe tracking is then executed with a sparsified number of ground truth keyframes (14, 9, 7, 5) and the camera pose is compared in each frame to the corresponding ground truth pose. In long-term use, even a small number of keyframes eventually outperforms the dense tracking due to drift. However, a small keyframe number produces local drift which appears as error ramps at keyframe switching points. This can be visually disturbing. With sufficient number of keyframes, the error remains small. Keyframe pose consistency finally depends on the on the accuracy of bundle adjustment. In this experiment, we obtain larger keyframe switching effects due to bundle adjustment phase. Bundle adjustment removes global error but can introduce local variance to the keyposes if the 2D point correspondencies are not precise. With the demonstration video, 14 keyframes produces small switching effects.

7.1. Rejecting dynamic foreground in studio

During broadcasting, the scene will contain moving actors. The pose estimation should only use static background in order to avoid estimation bias. After the static

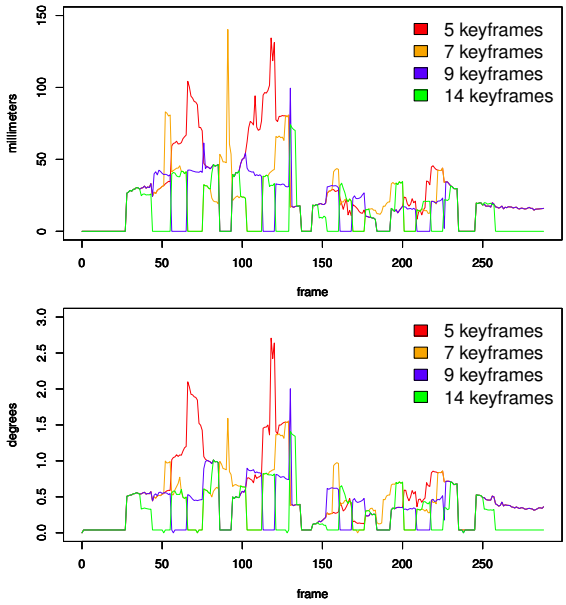

Figure 14: The translation and rotation error as a function of keyframe count. The sequence is illustrated in Figure 1. Keyframe switching error depends on the amount of keyframes. Over 14 keyframes are required to obtain sufficiently small keyframe switching error.

keyframe model has been captured, the scene can be segmented into static and dynamic components based on the current depth and intensity difference to the static keyframe. The weighting computed in section 3.3 enforces the estimation to use static background. The weighting is illustrated in Figure 15. When relying on incremental tracking (alg. 2), the reference RGB-D measurement may unfortunately contain also moving objects which interfere tracking. The effect can be noticed during the last cycles (Figure 13) where an actor is moving in the scene. The estimation requires sufficient amount of visible points and, therefore, foreground actors are not allowed to occlude more than a fraction of the selected points. Figure 13 illustrates how the moving actor does not disturb camera tracking when relying on keyframes.

7.2. *Studio lighting conditions with Microsoft Kinect*

In studio environments, the color constancy assumption works well, because lighting can be fixed and specular surfaces avoided. This implies that keyframe database is valid as long as the scene configuration is fixed. Interfering IR light which is not originated from the RGB-D sensor can easily be cut-off in studio environments. One problem, however, is that the Microsoft Kinect RGB camera is of low quality and in standard

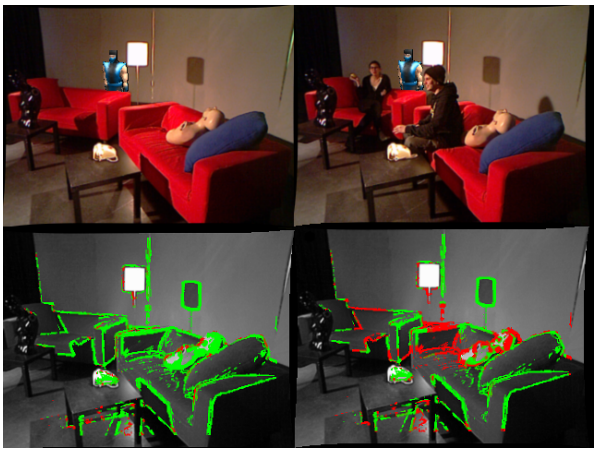

Figure 15: The points which currently participate in pose estimation are selected based on the image gradient magnitude. The dynamic foreground is given zero weighting in the estimation based on large depth and intensity difference to the static keyframe. The weight 1.0 is assigned to the green points and 0.0 to the red points.

lighting conditions the colors can saturate (see Fig. 16). With professional HD cameras this is not the case since they have remarkably better dynamic range. To circumvent this limitation, the RGB-D sensor depth maps should be directly calibrated with a HD camera [18].

Figure 16: HD camera (left) and Microsoft Kinect RGB image (right) in typical studio lighting conditions.

7.3. *Constraints*

If the camera is moving too quickly, the minimization may not converge, because a local optimization strategy is used. Global minimization strategies of the cost function are not discussed in this paper because they are computationally too expensive to operate in real-time. Motion capture systems, on the otherhand, are expensive but could be used to provide initial guess for a pose. To avoid re-localization needs completely, we operate with limited camera speed, sufficient scene texturing, sufficient keyframe density, and ensure that sufficient amount of selected keyframe points are visible despite the occluding actors.

8. Conclusions

In this work, an affordable real-time matchmoving solution has been developed, which can be used to produce broadcasts with interactive digital components, such as virtual characters and stage items. The solution performs in real-time using a RGB-D sensor, and a laptop with a low-end GPU. This system was able to accurately and robustly track camera in broadcasting studio sized operating volumes where the state-of-the-art approach fails. Time-evolving drift problem is solved by tracking the camera relative to the nearest keyframe. The keyframes were generated using incremental dense tracking, and fine-tuned using sparse bundle adjustment. M-estimator was enhanced by segmentation-based weights, which allows actors to move in the foreground while tracking the camera. When operating in the RGB-D sensor range, our pose estimation accuracy depends mostly on texturing, which is trivial to manipulate/add in studio environments. Therefore the proposed method is easily adaptable to different types of studio scene settings, whereas KinectFusion is constrained by the amount of geometrical variations in the scene. Camera tracking has been demonstrated in a real broadcast studio with and without dynamic components. Drift and keyframe switching errors have also been quantified. Future work will address the practical issues of how studio staff and cameramen can use this computer vision system in live broadcasts. We will calibrate RGB-D sensor with a professional HD camera for better image quality. Moreover, combination of the best properties of our approach and KinectFusion will be investigated.

9. Acknowledgements

We thank Heikki Ortamo and Jori Pölkki for their professional support in a TV brodcasting studio.

References

[1] T. Dobbert, Matchmoving: The Invisible Art of Camera Tracking, Sybex, 2005.

[2] G. Laboratorium für Informationstechnologie, Hannover, Voodoo camera tracker: A tool for the integration of virtual and real scenes (2012).

[3] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, A. Fitzgibbon, Kinectfusion: Real-time dense surface mapping and tracking, ISMAR (2011) 127–136.

[4] T. M. Tykkälä, C. Audras, A. I. Comport, Direct iterative closest point for real-time visual odometry, in: ICCV Workshop CVVT, 2011.

[5] K. Konolige, M. Agrawal, FrameSLAM: from bundle adjustment to realtime visual mappping, IEEE Trans. on Robotics 24 (2008) 1066–1077.

[6] G. Klein, D. Murray, Parallel tracking and mapping for small ar workspaces, Proceedings of the International Symposium on Mixed and Augmented Reality (ISMAR) (2007) 225–234.

[7] G. Silveira, E. Malis, P. Rives, An efficient direct approach to visual SLAM, IEEE Trans. on Robotics 24 (2008) 969–979.

[8] A. I. Comport, E. Malis, P. Rives, Accurate quadri-focal tracking for robust 3d visual odometry, in: IEEE Int. Conf. on Robotics and Automation, ICRA'07, Rome, Italy, 2007.

[9] S. Baker, I. Matthews, Lucas-kanade 20 years on: A unifying framework, Int. J. Comput. Vision 56 (3) (2004) 221–255.

[10] T. M. Tykkälä, A. I. Comport, A dense structure model for image based stereo SLAM, in: IEEE Int. Conf. on Robotics and Automation, 2011.

[11] F. Steinbrücker, J. Sturm, D. Cremers, Real-time visual odometry from dense RGB-D images, in: Workshop on Live Dense Reconstruction with Moving Cameras at the Intl. Conf. on Computer Vision (ICCV), 2011.

[12] R. Newcombe, S. Lovegrove, A. Davison, Dtam: Dense tracking and mapping in real-time, in: ICCV, Vol. 1, 2011.

[13] C. Audras, A. I. Comport, M. Meilland, P. Rives, Real-time dense RGB-D localisation and mapping, in: Australian Conference on Robotics and Automation, 2011.

[14] Y. Ma, S. Soatto, J. Kosecka, S. Sastry, An invitation to 3-D vision: from images to geometric models, Vol. 26 of Interdisciplinary applied mathematics, Springer, New York, 2004.

[15] M. Meilland, A. Comport, P. Rives, A spherical robot-centered representation for urban navigation, in: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 2010.

[16] V. Podlozhnyuk, Histogram calculation in CUDA, in: CUDA SDK, Nvidia Corporation, 2007.

[17] R. B. Sidje, Expokit: Software package for computing matrix exponentials, ACM - Transactions On Mathematical Software 24 (1) (1998) 130–156.

[18] C. Herrera, J. Kannala, J. Heikkila, Joint depth and color camera calibration with distortion correction, IEEE Trans. on PAMI 34 (10).

[19] J.-Y. Bouguet, Camera calibration toolbox for matlab, http://www.vision.caltech.edu/bouguetj/calib_doc.

[20] J. Sturm, S. Magnenat, N. Engelhard, F. Pomerleau, F. Colas, W. Burgard, D. Cremers, R. Siegwart, Towards a benchmark for RGB-D SLAM evaluation, in: Proc. of the RGB-D Workshop on Advanced Reasoning with Depth Cameras at Robotics: Science and Systems Conf., 2011.

[21] R. B. Rusu, S. Cousins, 3D is here: Point Cloud Library (PCL), in: IEEE International Conference on Robotics and Automation (ICRA), 2011.

[22] P. Henry, M. Krainin, E. Herbst, X. Ren, D. Fox, RGB-D mapping: Using Kinect-style depth cameras for dense 3D modeling of indoor environments, The International Journal of Robotics Research 31 (5) (2012) 647–663.

[23] H. Hirschmüller, Stereo processing by semi-global matching and mutual information, IEEE Trans. on PAMI 30 (2) (2008) 328–341.

[24] M. A. Lourakis, A. Argyros, SBA: A Software Package for Generic Sparse Bundle Adjustment, ACM Trans. Math. Software 36 (1) (2009) 1–30.