Johan Hedborg & Björn Johansson

Real time camera ego-motion compensation and lens undistortion on GPU

Abstract This paper describes a GPU implementation for simultaneous camera ego-motion compensation and lens undistortion. The main idea is to transform the image under an ego-motion constraint so that tracked points in the image, that are assumed to come from the ego-motion, maps as close as possible to their average position in time. The lens undistortion is computed simultaneously. We compare the performance with and without compensation using two measures; mean time difference and mean statistical background subtraction.

Keywords GPU · camera ego-motion compensation · lens undistortion

1 Introduction

Intersection accidents are over-represented in the statistics of traffic accidents. This applies to everything from fatal accidents to mere property damage. This problem has received greater attention by the Swedish Road Administration (SRA) in connection with its work on Vision Zero\(^1\). As part of the IVSS (Intelligent Vehicle Safety Systems) project 'Intersection accidents: analysis and prevention', video data is collected at intersections. The video data and other data (e.g. data collected using a test vehicle, and data collected from a driving simulator) will be used to test research hypotheses on driver behavior and traffic environment factors, as well as a basis for proposing ways to improve road safety at intersections.

Many intersections do not have buildings nearby, and cameras then have to be mounted on poles that sway due to the wind. Image analysis of video data, such as vehicle tracking, is easier if all motions in the video are caused by moving objects. The image sequence then has to be compensated for camera ego-motion. Moreover, the ego-motion compensation will also improve the accuracy of the camera calibration, i.e. the mapping between the 3D world and the 2D image plane. This will in turn improve the accuracy of image based 3D position estimation.

Camera lenses cause lens distortion that have to be taken into account when estimating vehicle positions, size, etc. from the image. One way to deal with this problem is to transform the image to an undistorted equivalent that does not contain lens distortion.

GPUs have successfully been used in a large variety of computationally intensive problems. It is not uncommon to see speedups of 10-100 times and for some special cases over 300. The GPUs' tight adaption to visualization make them especially well suited to many kinds of image analysis.

We will in this paper describe a theory and a GPU implementation for simultaneous compensation of camera ego-motion and lens undistortion. Figure 1 shows an overview of the algorithm.

The paper is outlined as follows: Sections 2-4 describe the theories for image point tracking, lens undistortion, and camera ego-motion estimation. Section 5 gives the implementational details. Section 6 gives a short description of a statistical background model estimator, which is used as a part of the performance evaluation experiments in section 7.

2 Image point tracking

Our camera ego-motion estimation needs a number of tracked points to compute the 'mean' image (point) position. For this we use the KLT (Kanade-Lucas) tracker, see e.g. (Shi and Tomasi 1994; Lucas and Kanade 1981; Tomasi and Kanade 1991), which tracks a local region template with a subpixel translation model. We initially intended to use the Harris interest point detector, see (Harris and Stephens 1988), to find suitable regions to track (see also (Shi and Tomasi 1994)). However, in our case the image contrast varied both globally (the

---

\(^1\) The policy on reduction of traffic accidents.
Sec 2: Image point tracking

KL T \Rightarrow \downarrow

Sec 3: Map points to undistorted domain

ω, s, c \Rightarrow \omega, s, c

Sec 4: Estimate camera ego-motion from current point positions to average point positions

\Rightarrow \Rightarrow t, \Omega

Simultaneously map distorted image to undistorted domain and to the mean position

Simultaneously map distorted image to undistorted domain and to the mean position

\omega, s, c \Rightarrow t, \Omega

Fig. 1 Process overview.

cameras automatic adaptation to varying illumination was limited and varied slowly) and locally (half the view could lie in the shadow of a building). These variations require globally, or even locally, time varying adjusted thresholds. Furthermore, the background was mostly of low contrast, and entering vehicles of high contrast would ‘steal’ many interest points if the interest point detector is not carefully designed, see figure 2 for an example.

Fig. 2 Example of scene where most high contrast interest points can be potentially occluded by vehicles.

We wanted to use a simple method that is still sufficient. We finally decided to use a regularly sampled grid, in order to have the points spread out in the entire image in a simple manner, see figure 3. In our experience, most features behaved well even in the fairly homogeneous regions (e.g. on the road or in the grass regions), and the remaining ones was detected by the outlier criteria, which is still needed even if more elaborate features were used.

Fig. 3 Example of point grid.

In practice, the point grid has to be reset once in a while due to changes in light and in the static environment. We therefore use two grids that overlap in time and are alternately reset. A first per-point bounding box outlier test is applied to eliminate points that are too far from their original position, e.g. are stuck on some moving vehicle. This step is necessary for getting a good initial guess for the ego-motion solver. This outlier detection rule is quite simple and general and eliminates the need for more complex methods such as RANSAC. The valid grid points are later fed in the ego-motion solver to estimate the motion of the camera.

3 Lens distortion

Lens undistortion deals with the mapping between the captured, radially distorted, image and an undistorted image where the pinhole camera model can be applied, see figure 4 for an example.

For our application we used the FOV model from (Devernay and Faugeras 2001), but the choice of model is not critical for the implementation performance due to the highly programmable nature and computational performance of a modern graphics card. The model is briefly described below.

Let \( x_d \) denote a point in the distorted image, \( x_u \) the corresponding point in the undistorted image, and \( r_d = ||x_d||, r_u = ||x_u|| \). The FOV model is defined as

\[
r_d = \frac{1}{\omega} \arctan \left( 2r_u \tan \frac{\omega}{2} \right) \Leftrightarrow r_u = \frac{\tan(r_d \omega)}{2 \tan \frac{\omega}{2}}.
\]

(1)

This model corresponds to an ideal fish-eye lens (but also seems to work well for more regular lenses, according to (Claus and Fitzgibbon 2005)). The parameter \( \omega \) corresponds to the field of view (FOV) angle.
We have observed that this model does not preserve the metric in the image center (i.e. \( \frac{dr}{dr_d} = 1 \) at \( r_d = 0 \)), which sometimes is desirable if we do not want to lose resolution anywhere in the image. Of course one could always rescale \( r_u \) in \( \omega \rangle \) after the mapping from \( r_d \), to fulfill the preservation requirement. But this is equivalent to using the simpler expression

\[
r_d = \frac{\arctan(r_u \omega)}{\omega} \Leftrightarrow r_u = \frac{\tan(r_d \omega)}{\omega}.
\]

Note that this new \( \omega \) (the same symbol is kept for simplicity) has the unit \([\text{radians/pixel}]\) if \( r_d, r_u \) are measured

In line with (Devernay and Faugeras 2001), we also include parameters for the origin and aspect ratio according to

\[
x_d = \begin{pmatrix} s_x & 0 \\ 0 & 1 \end{pmatrix} x'_d + c_d
\]

(hence we use \( r_d = ||x'_d|| \) in \( \omega \rangle \) instead). The estimation of the lens parameters \((\omega, s_x, c_d)\) is beyond the scope of this paper, but we use a method is similar to (Devernay and Faugeras 2001).

Undistorted points, \( x_u \), will from now on be denoted \( x = (x, y)^T \) for simplicity.

### 4 Camera ego-motion compensation

#### 4.1 Ego-Motion model

There exist several different camera ego-motion models, depending on the type of camera motion (e.g. only rotation) and the geometry of the 3D world (e.g. flat world). In our case we have camera rotation as well as translation, and we did not want to restrict ourselves to a flat world since many intersections are close to buildings and trees.

Let \( T = (T_X, T_Y, T_Z)^T \) denote the instantaneous translation and \( \Omega = (\Omega_X, \Omega_Y, \Omega_Z)^T \) the instantaneous rotation. Moreover, let \( f \) denote the camera focal length and \((X, Y, Z)\) denote the 3D point that corresponds to the point \( x = (x, y)^T \) in the image in a common coordinate system, see figure 5.

It is well known (see e.g. (Heeger and Jepson 1992)) that the instantaneous image motion \( v \) at point \( x \), assuming a pinhole camera, in the general case can be expressed as

\[
v(x) = \begin{pmatrix} f & 0 & -x \\ 0 & f & -y \end{pmatrix} \left( \frac{1}{Z(x)} T + \frac{1}{f} \Omega \otimes \begin{pmatrix} x \\ f \end{pmatrix} \right)
\]

\[
= \frac{1}{Z(x)} \begin{pmatrix} f & 0 & -x \\ 0 & f & -y \end{pmatrix} T + \left( \begin{pmatrix} -\frac{xu}{T} & (f + \frac{x^2}{T}) - y \\ -\frac{yu}{T} & \frac{x^2}{T} \end{pmatrix} \right) \Omega
\]

Unfortunately (and naturally) the translation term depends on the distance to the observed 3D point, \( Z \), which is unknown in our case. Various approximations to the first (the translation) term have been tested, e.g.

\[
0, \quad \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} t, \quad \begin{pmatrix} x \\ 0 \\ 0 \end{pmatrix} t, \quad \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} t.
\]

The best choice depends (by empirical studies) on the camera setup relative to the 3D ground, existing static objects in the view, etc. We currently use the third one from the left. The choice of model may not be critical, but the background model estimation and subtraction contained somewhat fewer artifacts when using the chosen model. The camera was mounted on a 20m high pole, which swayed a few decimeters at the top, so the translation will still correspond to several pixels. Since the background models are sensitive to translations by even a few pixels, it still makes sense to include a translation term.

#### 4.2 Model estimation

The main idea in the ego-motion compensation is to warp the image, under an ego-motion constraint, so that tracked points in the image (assumed to follow from the ego-motion) map as close as possible to their average position in time.
Let \( \{x(t)\} \) denote a set of \( K \) points that have been tracked for a period of time, and mapped to the undistorted domain (the tracking is done in the original distorted image). Before we use these points to estimate the camera ego-motion, we first remove outliers by the following criteria: A point that has moved too far from its initial position (the camera pole motion is bounded) will be classified as an outlier for a period of time. If the point after that time is inside the box again, then it is reinstated as an inlier, otherwise it is removed for another period of time. Typically, this handles points that are temporarily occluded by passing objects.

The average position in time is then computed for the remaining points, denoted \( \{\bar{x}_k\} \). Next we compute the motion of point \( k \) at time \( t \) as \( v_k(t) = x_k(t) - \bar{x}_k \).

The camera ego-motion parameters \( t, \Omega \) in (4-5) can then be computed as the solution to a least squares problem by collecting equations (4-5) from the set \( \{v_k(t), x_k(t)\} \) (excluding the outliers).

As a way to make the estimation more robust, new outliers are found as points that do not fulfill the estimated ego-motion (4-5) very well, i.e. have a large residual. The least squares problem is then solved again to give the final estimate.

Once the ego-motion parameters are estimated, we can warp the image accordingly to the 'average' position. The lens undistortion is done at the same time. In this way we avoid the additional blur caused by resampling the image twice.

### 5 GPU implementation

#### 5.1 Programing Interface

There exist several possible ways to implement algorithms on a GPU. One way is to use a Graphics API (Application Programing Interface) such as DirectX or OpenGL. These APIs have a large support from different software and hardware platforms, especially OpenGL which lets you develop applications on everything from small mobile devices and PlayStation 3 to any PC. There are however some limitations on flexibility. It can be very difficult if not impossible to make some algorithms efficient even if they are parallel in their nature. This is the main reason why two of the biggest graphics hardware vendors have come up with more versatile APIs. NVIDIA has released CUDA, and AMD has developed the Stream SDK. These are more flexible and can make use of special hardware functionality, such as shared memory between processors and random address writes. These APIs are however vendor specific, and demand newer generations of their Graphics boards. We have chosen to implement our algorithm in a Graphics API (DirectX), mainly for two reasons. It uses hardware and functionality specific to Graphics APIs such as the the rasterizer, bilinear interpolation, write to texture. Some of these parts are missing from the more general APIs. Secondly, as far as we can see, the KLT algorithm would not benefit from the extra functionalities of the new APIs. Because of the similarities between the Graphics APIs, techniques used here can easily be transferred to OpenGL.

#### 5.2 The KLT Tracker

The KLT tracker is based on the early work of Lucas and Kanade (Lucas and Kanade 1981), and was fully developed by Tomasi and Kanade (Tomasi and Kanade 1991). They define the dissimilarity \( \epsilon(d) \) between two local regions in two images (\( I \) and \( J \)) as:

\[
\epsilon(d) = \iint_W \left( I(x + \frac{d}{2}) - J(x - \frac{d}{2}) \right)^2 dx.
\]

Here \( d = (d_x, d_y) \) is the displacement between the two regions and \( W \) defines the spatial window (patch size). To find the displacement, the dissimilarity (6) is approximated by its first order Taylor expansion and the minimum is found by differentiation. For the case of discrete images the solution is computed from the \( 2 \times 2 \) equation system:

\[
\begin{align*}
\sum_{w} \left( \sum_{g_y} g_y^2 \cdot \sum_{g_x} g_x \cdot g_y \right) & \left( \frac{d_x}{d_y} \right) \\
= 2 \left( \sum_{w} \left( \sum_{g_x} (I - J) g_x \right) \right),
\end{align*}
\]

where \( g_x = I_x' + J_x' \) and \( g_y = I_y' + J_y' \). A more detailed derivation is found in (Hedborg et al 2007).

#### 5.2.1 Calculation Scheme

The KLT algorithm is an iterative process, where each iteration can be divided into 5 steps, see Figure 6. Here follows a short explanation of each step. We first extract the two patches (linearly interpolated) that are going to be matched, for the following steps only the elementwise sums and differences are needed so these are also calculated here. Next we apply an neighboring pixel difference filter to \( I_w + J_w \). The gradients are multiplied elementwise, as shown in the figure. Then we sum over the 5

![Fig. 6 Illustration of the different steps in a GPU implementation of a KLT tracker.](image_url)
unique elements. The last step of the iteration is to estimate the disparity by solving the 2x2 equation system (7). The J patch position is adjusted according to the current disparity estimate and then the iteration process is started over again. Several different stop criteria exist, this implementation uses a fixed number of iterations.

The KLT has been implemented on the GPU before (Sinha et al 2006). There is however a significant difference between their implementation and ours. They do all the calculations for an entire patch and all its iterations in one shader thread (corresponds to one pixel calculation when doing graphics on the GPU). This is a good solution for the problem if there are enough points to track, however if there are a smaller amount of points (as in our implementation) the GPU will not be fully utilized. The main reason for this is that the GPU uses a large amount of threads to hide memory load latency and thread synchronization. Our method divides the patch calculations to a larger extent between different processors, and therefore can handle smaller amounts of patches more efficient (and equally well for larger amounts).

5.3 Performance

The hardware that was used for measuring the performance of the implementation has an Intel Core 2 Duo E6600 CPU, and the graphics board used is based on the Geforce 8800 GTX GPU from NVIDIA. We have divided the GPU Performance evaluation into the two main parts, the KLT and a lens distortion part. When evaluating the KLT we compared it with the freely available GPU-KLT tracker (Sinha et al 2006) This tracker is partly reconfigurable and was set to 31x31 patches in one scale. However it was not trivial to convert it to a color tracker so this was not done. The image size is 1024x768. As mentioned before the two trackers have a very different design, this tracker issues one thread per patch while our issues 32x32 threads per patch. A GPU must have a large amount of threads to have full occupancy of all processors, often thousands (Kirk and W. Hwu 2007). In our implementation only 256 patches are calculated simultaneously and will therefore benefit greatly from our design of the KLT tracker. Table 1 shows the performance between the two trackers for 256 and for 1024 simultaneously tracked patches. This shows the scaling advantage for using our approach to increase the number of threads in the implementation.

<table>
<thead>
<tr>
<th>Tracked Patches</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our (color)</td>
<td>8.7 ms</td>
<td>28 ms</td>
</tr>
<tr>
<td>(Sinha et al 2006) (grayscale)</td>
<td>26 ms</td>
<td>44 ms</td>
</tr>
</tbody>
</table>

The lens distortion algorithm is very well suited to the GPU, it can use the linear interpolation hardware and the texture cache’s advanced prefetch mechanisms which are available on a modern GPU. The CPU implementation utilizes a well known Computer C-library called OpenCV. OpenCV’s cvRemap() command was used with a predefined lookup table and two floating point coordinates for each pixel. On the GPU the distortion mapping was calculated online. The size of the input image was 1024x768 and the output image was 1440x1050, all pixels where sampled using linear interpolation. These kinds of calculations are near optimal for the GPU and a speedup of over 250 times was achieved. The performance numbers are shown in table 2.

Table 2 Lens undistortion/rectification performance

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (OpenCV)</td>
<td>115 ms</td>
</tr>
<tr>
<td>GPU</td>
<td>0.45 ms</td>
</tr>
</tbody>
</table>

When solving the 5 parameter least square problem for the ego-motion model in section 4.2, a CPU version of Lapack is used. The LSQ-problem is small and fits easily in the cache memory of the CPU, when this is true and an optimized Lapack routine is used, the performance will often be on a pair with a GPU implementation. Under the assumption that the routine is as fast as a GPU solution, it is a lot easier to use an already existing CPU-implementation.

6 Statistical background subtraction

We will in the experiments use time difference, \(|I_u(t) – I_u(t-1)|\), to evaluate the stabilization performance. However, the subsequent step in an image processing based object tracking system with (almost) static cameras is usually some background estimation and subtraction. One method for background subtraction that has become popular in recent years is (Stauffer and Grimson 1999) statistical background modeling. This method collects color statistics in each pixel individually during time and detects foreground pixels as pixels that have an uncommon color. This method has also been combined with shadow detection in (Wood 2007), and this combination is currently used in our system. The statistical background model has some ability to handle ego-motion due to the statistical model, but we will show in the experiments that the compensation still improves the result.

We currently use a CPU implementation of this method, but it can also be implemented in GPU, see e.g. (Lee and Jeong 2006).
7 Experiments

Figures 7-13 show some results on a windy day. The used video was 1h long at 20 fps, but the plot shows only a typical portion. The plot in figure 7 shows the average time difference, i.e. \( \text{mean}(\| \mathbf{I}_u(t) - \mathbf{I}_u(t-1) \| > \text{threshold}) \), as function of time. We choose to compute the average after thresholding, to ignore the effects from sensor and compression noise that otherwise gives a large total contribution even though it is low in each pixel. The plot shows the average both without and with ego-motion compensation. We see that the average without the compensation contains a lot of spikes, which are to a great extent removed by the compensation. The remaining fluctuation mainly comes from moving objects.

The plot in figure 8 shows the corresponding average for the statistical background subtraction with shadows detected and removed. This method can deal with some ego-motion even without the compensation, but the result is still better after the compensation.

Figures 9-12 shows the frames where the averages with and without compensation differ the most, i.e. in a sense the 'best' and 'worst' cases. However, the 'worst' case only shows that the performance is not decreased when using the compensation. There are still some cases in the sequence where the compensation has been insufficient, due to some extraordinary motion, figure 13 shows an example. These cases however appear to be quite rare.

8 Performance

For measuring the overall performance of the implementation, see table 3.

<table>
<thead>
<tr>
<th>Patch Size</th>
<th>8x8</th>
<th>16x16</th>
<th>32x32</th>
<th>64x64</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>30</td>
</tr>
<tr>
<td>256</td>
<td>60</td>
<td>50</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>512</td>
<td>60</td>
<td>45</td>
<td>30</td>
<td>12</td>
</tr>
</tbody>
</table>

We choose as setting 512 (2 layers, each 16×16) 32×32 patches (shown as boldface in the table) which gave us a total time of 33 ms/frame, which for our application is well within the bounds of a real-time system. A significant part of the total time per frames was spent on the KLT-tracker, two grids of 256 points each take 17.4 ms of the total 33 ms. If we had used a CPU version of the lens undistortion and rectification part, it would have taken over 100 ms and the performance of the system had been far from what it is now. The performance advantage when using GPU in a KLT implementation is shown in (Sinha et al. 2006), and in this setup our tracker clearly outperforms theirs.

9 Conclusions

On-line processing at real-time is necessary to cope with large streams of data generated from surveillance. To do both stabilization and undistortion of a video stream in real-time for large images is not possible on current generation of CPUs, mainly due to the large amount of data.
Fig. 9 The frame where the averages with and without compensation has the largest negative difference. Col 1: Undistorted image. Col 2: Frame difference without ego-motion compensation (using th=50, the original had 255 as max value on each color channel). Col 3: Frame difference with ego-motion compensation.

Fig. 10 The frame where the averages with and without compensation has the largest positive difference.

Fig. 11 The frame where the averages with and without compensation has the largest negative difference. Col 1: Undistorted image. Col 2: Statistical subtraction without ego-motion compensation. Col 3: Statistical subtraction with ego-motion compensation.

Fig. 12 The frame where the averages with and without compensation has the largest positive difference. (Note that the subtraction indicates a false vehicle near the center of the image. This is because a vehicle earlier in time has been temporarily stationary for a longer period of time and has therefore become part of the background.)
that has to be processed. The presented algorithm for simultaneous lens undistortion and ego-motion compensation is sufficiently fast to run in real-time and seems to work well in most cases, especially when combined with a statistical background model that takes care of the remaining ego-motion.

Acknowledgements This work has been supported by the Swedish Road Administration (SRA) within the IVSS project Intersection accidents: analysis and prevention.

References

Shi J, Tomasi C (1994) Good features to track. IEEE Conference on Computer Vision and Pattern Recognition pp 593–600