Cimarron

Stabilisation of videos in modern C++

Marius Herget

in partnership with

Institute for Computer Science
Ludwig-Maximilians-Universität München

12th October 2018

1 Idea

Video stabilization is used ever since cameras evolved. In the early days, physical stabilization techniques as tripods were used. In the following centuries cameras enhanced step by step. New solid and dynamic methods were invented like steady cams, dollies, shoulder rigs and many more. With the invention of digital photography and videos, another possible solution was found: digital image stabilization. Different techniques like optical flow analysis or warp stabilization were developed. `Cimarron` implements such a feature tracking method for motion compensation.

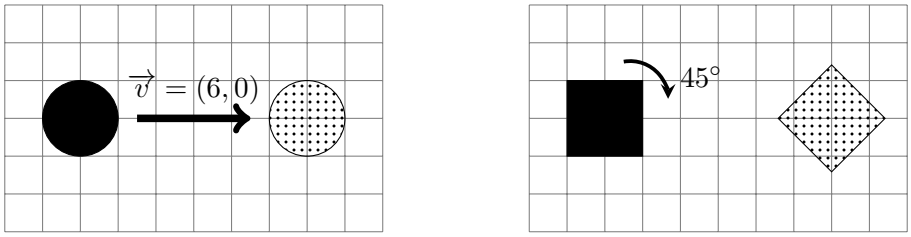
2 Theoratical introduction

New technologies emerge each year. In the last years, especially phones and small cameras were published. Under ideal condition, recent smartphone's cameras pictures cannot be distinguished from professional cameras anymore. Nevertheless, a smartphone video is often detectable by its *handheld*, shaky look. As already mentioned within the short introduction different methods can be used to compensate this motion.

The general idea of video stabilization is to counter, smoothen or to minimize unwanted shakes. In general video, motion stabilization can be classified into three categories: mechanical based, optical based and electronically based. Instead of using specific hardware like the first two methods, the electronic approach uses computing power to implement image processing techniques in the postproduction step. [4]

In order to compensate for the unwanted movement of the camera, motion can be described in various forms. *Translation* is the simplest form of expression. In this concept direct, linear movement of a single point is described as the distance it covered within a certain time. This can be enhanced with the combination of *rotational motion*. In comparison to the translational movement, it specifies the angle a point/body covers in a given timeframe. Examples can be seen in fig. 1.

Another motion model is *perspective*. As well as the translational and rotational movement it can be described as vectors and scalars. The inverse of each model can be used as the source for the stabilization.



(a) Translational motion (b) Rotational motion

Figure 1: Differnt motion models

3 Modeling

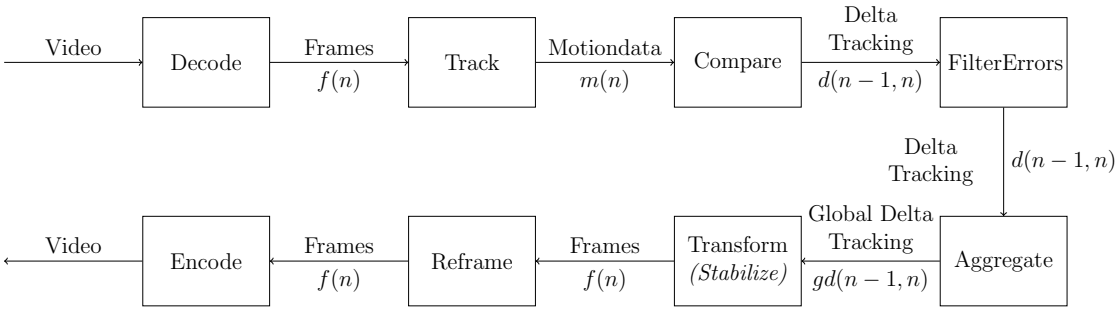


Figure 2: High-level system diagram

The system diagram shown in fig. 2 concludes to three different simple concepts:

Frame n-dimensional matrix

The frame is used to contain all data for an image. It allows access to the individual pixels, dimensions and some functionality. An object which fulfills this concept can be passed on to every image processing function. The concept can also be found in OpenCV's `Mat` class.

TrackingWindow rotating rectangle

The TrackingWindow concept allows to specify an area to visualize and save a tracking area. It saves information about the center point, its size and the rotation.

FrameDeltaImage vector of delta in two frames

The FrameDeltaImage is the resulting data from comparing two images. It

saves information about the two frames and its differences in the tracked objects.

Expression	Semantics	Equivalent expression

4 Implementation

`Cimarron` is implement in `C++14` and heavily depends on the `OpenCV` library. Figure 2 shows the general structure of the application. The different modules each describe a specific step to achieve a smooth video. The first step is to decode the input video and to extract each frame. Therefore, it uses the *frame* concept described earlier. *Track* is the implementation of the OpenCV Continuously Adaptive Meanshift (CAMshift) algorithm, which is an improved version of the *Meanshift* algorithm. It generates `motionData` of the tracked objects which are then compared, filtered and aggregated to achieve one global difference vector for each frame of the video. In the end, these vectors are used to transform the specific frames to compensate the shaky movement. In the end, these modified frames are reframed with a simple, non-dynamic mask and encoded in an *avi* video file. `Decoding` and `Encoding` are trivial due to the use of the `video++` framework and are not discussed in this documentation. More information on these topics can be found in [2, 3].

4.1 Feature tracking

The general concept of `Cimarron` is to track objects within the frames and follow those throughout the frames. Therefore, the CAMshift algorithm is the best way to implement tracking. It is based on the Meanshift algorithm, which uses the simple approach of finding the maximum density region of points in a given search window. This data can be a pixel distribution like histogram back projection.

Firstly, Meanshift analyses the initial search window and calculates the centroid of the data. Afterward, it uses this centroid as the new center of the search window and repeats this process, until the center of the area and the newly calculated centroid are within a margin of error.

One disadvantage of Meanshift which is addressed by CAMshift ist that the tracking area has always the same size. Therefore, it is not possible to address moving objects which change their size. CAMshift was published by Gary Bradsky in his paper "Computer Vision Face Tracking for Use in a Perceptual User Interface" in 1998 and works in the following steps:

1. Apply Meanshift until it converges.
2. Updates the size of the search window.
3. Update the rotation of the search area.
4. Repeat step 1. until the required accuracy is met.

A detailed explanation with specific mathematical formulas can be found in [1].

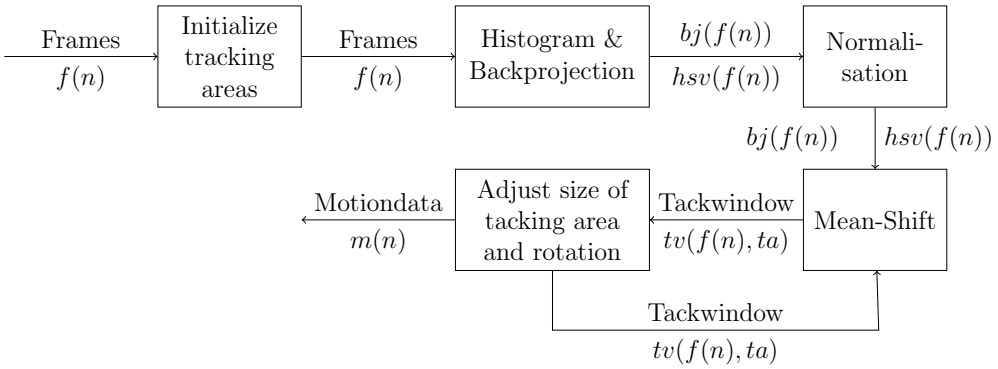


Figure 3: Detailed system diagram of `Track`

Figure 3 shows the implementation of the CAMshift algorithm in `Cimarron`. The nine initial tracking areas are ordered with a 3×3 Grid in the frame and can be seen as the red rectangles in fig. 4. Each tracking area is independent and uses special `camShiftTracker` class.

The second step is to prepare each frame within the tracker. Therefore, a back projection is used. This method uses the histogram of the initial tracking area in an image to show up probabilities of colors may appear in each pixel [5]. The steps of creating such this are shown in listing 1:

1. Transform image in an color space which saves the *hue* of each pixel.
2. Extract the hue information to receive a grayscale image and normalize its histogram.

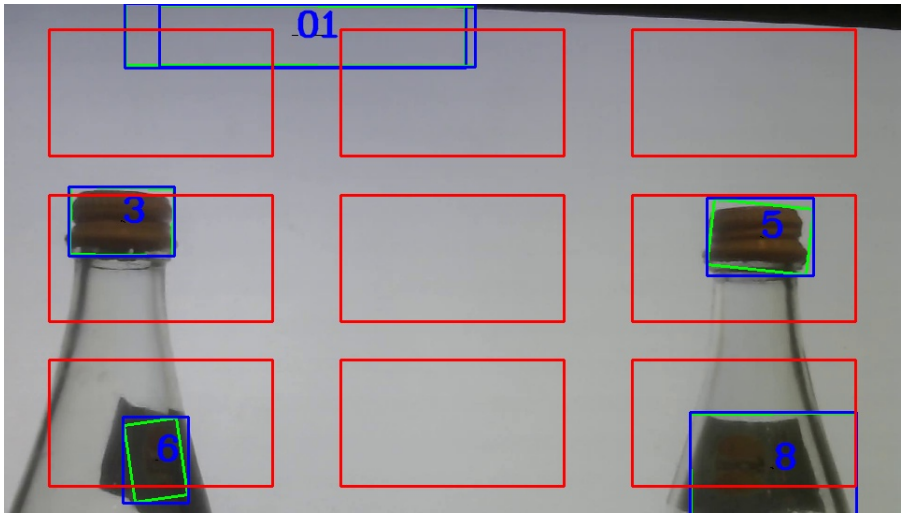


Figure 4: Example frame: tracking

3. Use OpenCV's `calcBackProject`.

Listing 1: Preparation for tracking

```
cv::cvtColor(image, hsv, CV_BGR2HSV);

cv::inRange(hsv, cv::Scalar(0, _smin, MIN(_vmin, _vmax)),
        cv::Scalar(180, 256, MAX(_vmin, _vmax)), mask);
int ch[] = {0, 0};
hue.create(hsv.size(), hsv.depth());
cv::mixChannels(&hsv, 1, &hue, 1, ch, 1);

if (_start) {
    cv::Mat roi(hue, _selection), maskroi(mask, _selection);
    cv::calcHist(&roi, 1, 0, maskroi, hist, 1, &_hsize, &phranges);
    cv::normalize(hist, hist, 0, 255, CV_MINMAX);
    _trackWindow = _selection;
    _start = false;
}
```

The result is an color-weighted grayscale projection of the image. `calcBackProject()` functions as follows: (i) Calculate weight of each color by the histogram and (ii) Multiply each color of each pixel with its weight.

From there OpenCV takes it for us by simply calling `cv::CamShift` with the created back projection, the tacking area and a terminating variable.

Listing 2: CAMshift call

```
cv::CamShift(
    backproj, _trackWindow,
    cv::TermCriteria(CV_TERMCRIT_EPS | CV_TERMCRIT_ITER, 10, 1));
```

For each frame and tracking area, this process is run through. The result can be seen in fig. 4. The green rectangles are the tracked elements, the blue ones their bounding rectangles. Small lines indicate their movement so far through the frames.

4.2 Movement identification of tracked objects

The feature tracking returns a `std::vector<motionVector>` aka `motionData`. Whereby, `motionVector` is a struct of all tracking vectors in the frame and its index. This data needs to be transformed into difference vectors between frames. These results should imply how the object moved between two frames.

This is simply achieved by comparing each tracking vector by its corresponding one in the previous frame:

$$\Delta_{TrackingVector_i(m(n))} = TrackingVector_i(m(n-1)) - TrackingVector_i(m(n))$$

This results in the difference of following properties (i) $\delta CenterPoint$, (ii) $\delta Angle$ and (iii) $\delta Area$. These results are combined to a frame delta data vector.

The next step is to filter tracking errors by simply looking up delta vectors which exceed a threshold of over 10% movement between two frames and deleting those.

Since the goal is to counter shaky movement by its translational and rotational movement, it is important to recognize the intentional movement of objects within the frame and shakes of the camera. To achieve this, an aggregation of the delta vectors needs to be done. The exceptions are: 1. There are tracked objects in the image and 2. there is only one tracked object. The first case returns the single delta vector as the global delta vector. The second one returns a zero vector which indicates that there is no movement between the images. The general case is that there are multiple motion vectors in each frame and therefore, multiple delta vectors.

In this case, each vector is compared on time to each other vector and its similarity in translational and rotational movement is calculated.

Vector similarity is computed by the cosine similarity:

$$cos\_sim = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

It shows the the angle between two vector by calculating the dot product of each member of the vector and dividing it by the multiplication of the vector's magnitude. A result of 1 indicates that the vectors are the same. Otherwise, a result below 0 shows that the vectors are dissimiliar and an exact 0 means they are orthogonal.

Listing 3: Cosine similarity

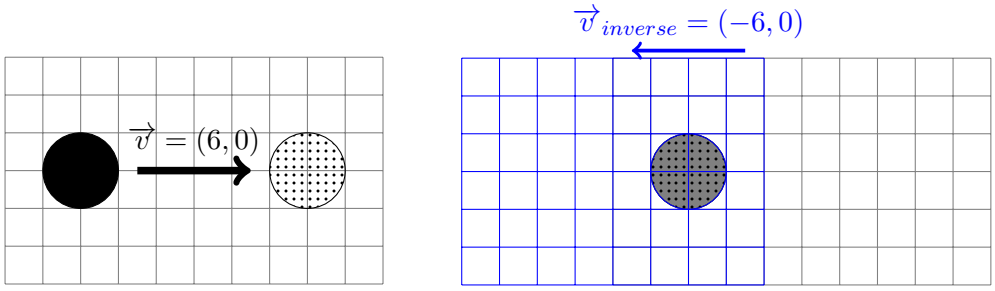
```
float dot = 0.0, denom_a = 0.0, denom_b = 0.0;

dot += A->deltaPosition.x * B->deltaPosition.x +
       A->deltaPosition.y * B->deltaPosition.y;
denom_a +=
    std::pow(A->deltaPosition.x, 2.0) *
    std::pow(A->deltaPosition.y, 2.0);
denom_b +=
    std::pow(B->deltaPosition.x, 2.0) *
    std::pow(B->deltaPosition.y, 2.0);
auto ret = dot / (std::sqrt(denom_a) * std::sqrt(denom_b));
```

Angle similarity is simply calculated by the percentage difference between the two angles.

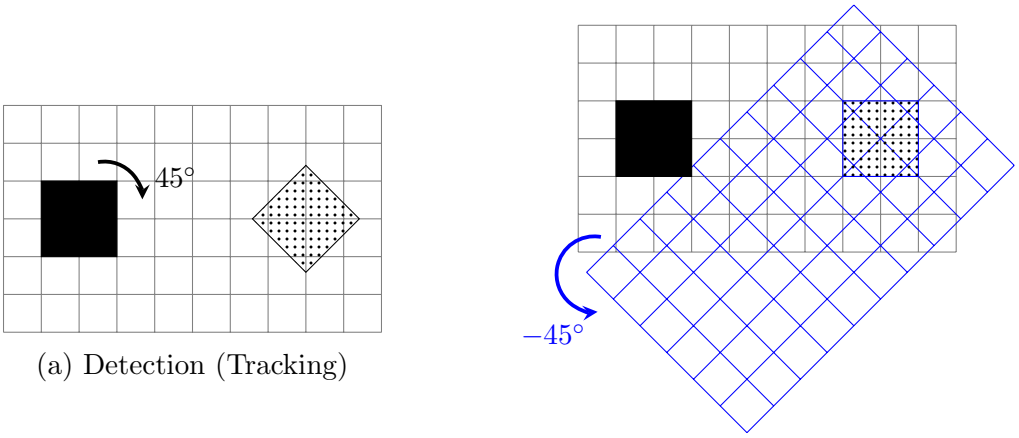
In the end, the analysis checks whether there are enough delta tracking vectors which are similar:

1. Count translational and rotational data which exceeds the custom threshold of similarity (Vectors: > 0.7, Angles: $< 10\%$).
2. If at least 75% of all combinations exceed the similarity threshold, calculate their average value. Otherwise, the delta is zero.
3. Create a global delta vector with the computed delta data.



(a) Detection (Tracking) (b) Inverse

Figure 5: Translational compensation



(a) Detection (Tracking)

(b) Inverse

Figure 6: Rotational compensation

4.3 Stabilization

As already described in section 2 the inverse can be used to compensate movement based on the moving vector. Figures 5 and 6 show how it can be accomplish.

The implementation is thanks to OpenCV quite simple. Firstly, each frame is resized to:

$$(width_{new}, height_{new}) = \\ (width_{original} + max(globalDeltaData_x), \\ height_{original} + max(globalDeltaData_y))$$

The translational and rotational transformations can be seen in listing 5. `cv::Mat` fulfills the frame concept and only differs in its memory management. One flaw in the current implementation is that the image is rotated around its center. A better solution would be calculation the middle of all relevant tracking areas and using this point as the center for the rotation.

Listing 4: Frame transformations

```
void positionShift(cv::Mat &image, frameDeltaVector dv) {
// Perform position shift
cv::Mat trans_mat = (cv::Mat_<double>(2, 3) << 1, 0,
                     dv.deltaPosition.x, 0,
                     1, dv.deltaPosition.y);
cv::warpAffine(image, image, trans_mat, image.size());
}
void rotate(cv::Mat &image, frameDeltaVector dv) {
// Perform rotation
cv::Point2f center((image.cols - 1) / 2.0,
                   (image.rows - 1) / 2.0);
cv::Mat rot = cv::getRotationMatrix2D(center,
                                      dv.deltaAngle, 1.0);

cv::warpAffine(image, image, rot, image.size());
}
```

4.4 Reframing

The last step of `Cimarron` is a simple but effective reframing. This is necessary since The frames will be repositioned and therefore leave some areas black. For aesthetic reasons a simple reframing with a mask is implemented which are shown in **??**.

Listing 5: Frame transformations

```
decltype(auto) mask(framevector const &_f, globalDeltaData const &_gdd) {
  auto resize = calcMaxTransformation();
  auto ffirst = clone(_f[0], _border = 0);
  cv::Mat mask =
      cv::Mat::zeros(to_opencv(ffirst).size(), to_opencv(ffirst).type());
  for (int c = 0; c < mask.cols; c++)
```

```
    for (int r = 0; r < mask.rows; r++)
      if ((r > resize.y * 2 && r < mask.rows - resize.y * 2) &&
          (c > resize.x * 2 && c < mask.cols - resize.x * 2))
        mask.at<cv::Vec3b>(cv::Point(c, r)) = cv::Vec3b{255, 255, 255};

  framevector reframed;
  for (auto frame : _f) {
    auto freframe = clone(frame, _border = 0);
    auto freframec = to_opencv(freframe);
    cv::Mat out;
    freframec.copyTo(out, mask);
    reframed.push_back(vpp::from_opencv<vuchar3>(out));
  }
  return reframed;
}
```

The method takes the maximum positional change in the stabilization and adds a black border to the images. This is established with a mask consisting of an inner white rectangle which has the following size:

$s = (width_{stabilized} - max(globalDeltaData_x), height_{stabilized} - max(globalDeltaData_y))$

Everything out of this area is masked out.

4.5 Streaming

In prepatation for potential pipelining and multi threading `Cimarron` implements a simple streaming semantics which results in the following semantics:

Listing 6: Cimarron semantics

```
auto vo = inputfile >>
          cimarron::pre::preprocessing(100) >>
          cimarron::analysis::analysis() >>
          cimarron::stabilziation::stabilziation({'p', 'r'}) >>
          cimarron::post::reframing() >>
          cimarron::post::video_output(outputfilename);
vo.flush();
```

(a) Rawframes

(b) Stabilzed and reframed

Figure 7: Results (Frames $0 - 4$)

Whereby, `preprocessing(100)` implies to only use the first 101 frames and `stabilziation('p', 'r')` shows that positional and rotational stabilization should be executed. Both variables are optional.

5 Results and future work

The results shown in fig. 7b are quite disappointing. The frames are repositioned and the angles are corrected but it's not finetuned yet. Nevertheless, the application is fully working. The implementation has a high potential and its structure allow easy and fast enhancements.

There are multiple optimizations which should be done in the future. The stabilization is currently really harsh. The jumps between the frames to compensate the motions are too big. This can be optimized by enhancing the feature tracking and or by smoothing out the global delta vectors. In addition, the reframing is not optimal. A dynamic solution computing the pixels which are always filled with a frame and resizing to this area could be a great optimization. Another method is presented in [4] where the authors introducing an edge completion method to fill up empty pixels after stabilization.

References

[1] Gary R. Bradski. *Computer Vision Face Tracking For Use in a Perceptual User Interface*. 1998.

[2] M. Garrigues and A. Manzanera. "Video++, a modern image and video processing C++ framework". In: *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*. 2014, pp. 1–6. DOI: 10.1109/DASIP.2014.7115639.

[3] Matthieu Garrigues. *matt-42/vpp: Video++, a C++14 high performance video and image processing library.* https://github.com/matt-42/vpp. (Accessed on 10/12/2018).

[4] Chongwu Tang et al. *A fast video stabilization algorithm based on block matching and edge completion*. Oct. 2011.

[5] Eric Yuan. *OpenCV, meanShift, camShift.* http://eric-yuan.me/continuously-adaptive-shift/. (Accessed on 10/12/2018). 2013.