Quick Guide to the Pointing Gesture Recognition

Roland Kehl
rkehl@vision.ee.ethz.ch
Computer Vision Lab, Swiss Federal Institute of Technology, Zürich

Abstract

This paper describes the pointing gesture recognition component of the BlueC pipeline. It is an intermediate result of my research in the field of Human Motion Recognition (HuMoR). This paper should act as an easy manual for users of the GR library. To be able to understand the many tuning options and the side effects that may appear during runtime, it describes also the algorithm and therefore the key ideas behind the pointing gesture recognition.

1 Introduction

BlueC provides an ideal environment for gesture recognition applications since sixteen static cameras see the user from an almost complete set of viewing angles. Therefore, gesture recognition was one of the goals for the first phase of BlueC.

The main problem of gesture recognition is the dimensionality of the pose estimation problem. Fast tracking of articulated structures is far from trivial, due to the multitude of local minima for typical cost functions. Also, the search space is huge, due to the high number of degrees of freedoms. One solution to smaller the search space is to base the method on the silhouettes that are extracted for the construction of the visual hull. Therefore, motivated by [3] we decided to limit the set of gestures to a simple pointing gesture in the current state in order to use it as an additional tracking device.

Our pointing gesture is modelled as the line of sight connecting the eyes of the user and the fingertip of the pointing arm as seen in figure 1.

Therefore, we are searching for the positions of the eyes and the pointing hand in the world frame. If we assume that the user is standing upright in the cave while pointing to some location on the screen,
we can expect the eyes position to be the position of the head, or more simple, the position of the top of the head, minus some distance on the body axis towards the ground. Since both the top of the head and the pointing hand are quite outstanding points on the silhouettes in most views, we should be able to reconstruct their positions in the world frame as soon as we have detected them on the silhouettes and have solved the correspondence problem between the views.

2 Algorithm

To understand all the switches and tuning possibilities in the main configuration file, we have to understand the basic algorithm first.

The basic algorithm can be outlined as follows:

1. Search for extremal points on the silhouettes, such as the head or the hand.

2. Solve correspondences between the different camera views for the previously detected points and reconstruct them with respect to the world coordinate frame.

3. Search for head positions in the previously found 3D points.

4. Search for on or two hand positions in the previously found 3D points.

2.1 Extremal points search

Similar to [2], we are looking for extremal points on the silhouettes. The idea is to detect the local maxima of the distance function $d_j(t)$ described by the distances $d_{ji}$ of every point $P_i$ lying on the silhouette $S_j$ to the centre of gravity $CoG$ of $S_j$ as shown in Figure 2(a). Figure 2(b) shows a typical distance function for a silhouette. The blue line indicates that the original distance function has been doubled. This is necessary since the marching squares algorithm uses a scan line algorithm to find a starting point for the silhouettes and therefore the top of the head is often the starting point of the silhouette. To improve the quality of the local maxima detection an FFT or box filter is used to smooth the distance function. The smoothed function with its local maxima and the corresponding extremal points on the silhouette are shown in Figure 2(c) resp. 2(d).
2 Algorithm

2.2 Solving correspondences

Since we already have computed the internal and external camera parameters with the self-calibration we use epipolar geometry to solve the N-view correspondence problem between the views.

Keeping in mind that we are dealing with rather small sets of points (typically 5-10) whose resulting reconstructed points will be separated quite nicely, we can split the problem down to two basic steps:

1. Solve correspondences for all triplet combinations of views.
2. Rise these correspondences up to get N-view correspondences.

2.2.1 3-View correspondence problem

To solve the correspondences between three views we use the trifocal constraint [1]: if three points in three different views are corresponding, then the intersection point of the epipolar lines of two points in the third view will correspond to the third point. This must be true for the epipolar lines in all three views. Figure 3 illustrates this epipolar constraint.

The idea is to check every possible triplet of points whether it fulfils the trifocal constraint or not. It is obvious that our measured points will not fulfil this constraint in most cases, even if they are corresponding, because we have quite noisy measurements. Therefore, it is more robust to choose randomly two points \( P_i, P_j \) from two views and search for the point \( P_k \) in the third view which is closest to the intersection point \( I_{i,j} \) of their epipolar lines. Then we give some tolerance \( t \) to the distance \( d_{i,j}^k \) from the closest point \( P_k \) to the intersection point \( I_{i,j} \) and mark the triplet as corresponding if \( d_{i,j}^k < t \).

At the end we get a set of triplets of points that seem to be corresponding. Every triplet has a distance \( d_{tot,i,j,k} \) associated with it which is the sum of all \( d_{i,j}^k \) from all combinations of the three views. As a last step we discard all triplet whose distance \( d_{tot,i,j,k} > t_{max}^{3view} \)

Figure 2: Extremal points search

![Figure 2: Extremal points search](image)
2.2.2 N-View correspondence problem

The goal is to merge all sets of 3-view correspondences to a single set of N-view correspondences by maximizing the likelihood of a correct match. This is done by a combinatorial use of the previously computed correspondence triplets and by taking into account the likelihood of a correct match for these triplets. We use the previously computed distance \(d_{tot}^{k,i,j}\) as the error measurement for a single triplet.

The algorithm does the correspondence solving and labelling in one step. The basic idea is to hand out the labels to the points so that all points belonging to a label will contain a minimal summed distance of all \(d_{tot}^{k,i,j}\) from the triplets which have been taken into account to generate the N-view correspondence.

As mentioned before, we are dealing with rather small sets of points whose resulting reconstructed points will be separated quite nicely. Together with the fact that we have sorted out some wrong correspondences previously in the 3-view step, we come to a robust algorithm that delivers a complete and correct set of N-view correspondences for our problem.

After we have computed the N-view correspondences, we can reconstruct the 3D points with respect to the world frame.

2.3 Head position search

It is obvious that the top of the head will be a quite outstanding point on the silhouettes of most views. Therefore, the top of the head point will be included in the set of reconstructed points from step 2.2.2 in most cases. So the problem comes down to the one of finding the top of the head point in the set of reconstructed points. Assuming that the user is standing upright while performing the pointing gesture, three simple constraints lead to our desired point:

1. The top of the head point will be above the shoulders, i.e. it will have a Z coordinate higher than a certain value (threshold \(t_{shoulders}\)).
2. If an overhead camera is available, then the projection of the top of the head point into the overhead camera view must lie inside the silhouette.
3. The top of the head point will be close to the main body axis, i.e. close to the axis parallel to the Z axis through the reconstructed centre of gravity.
Therefore, we sort out all points whose projection in the overhead view, if available, lies outside the silhouette and those with a Z coordinate $P_z < t_{\text{shoulders}}$. If more than one point survives (for example when the user rises the hand above the head), we apply the third constraint: we choose the point with is closest to the main body axis. Figure 4 illustrates these three constraints.

![Figure 4: Constraints used for head detection.](image)

Furthermore, we compute a Kalman prediction [4] of the previously detected head position for every frame. The prediction smoothes the noisy measurement and prevents the detected pointing gesture of nasty shaking.

If no point passes the validation step the whole algorithm will stop because no pointing gesture can been detected without the head position.

At the end, we just have to subtract some distance $d_{\text{eyes}}$ from the top of the head points towards the ground and finally get the position of the eyes $P_{\text{eyes}}$.

2.4 Hand position search

Our algorithm was originally thought to take advantage of an overhead camera. A first implementation lead to a robust algorithm for the hand detection by extracting 2D features from the overhead camera silhouette. Since not every environment will make it possible to mount an overhead camera we enhanced our pointing gesture recognition with a hand detection algorithm that is independent of an overhead camera.

The difference between the two algorithms is the way how they compute the 3D coordinates of the hands as described in 2.4.1 and 2.4.2.

2.4.1 Algorithm 1 - Using the overhead camera

This algorithm simply tries to detect the hands in the overhead silhouette and tries to solve the correspondences for these points to the frontal cameras.

It is obvious that the silhouette from an overhead camera will show the pointing arm quite nicely and therefore the pointing hand will be an extremal point in it. So we have to search for an extremal point on the silhouette that satisfies certain constraints. The basic algorithm can be outlined as follows:

1. Compute the centre of gravity $\text{CoG}$ from the overhead silhouette $S_o$.
2. Search for extremal points on the overhead silhouette as described in 2.1.
3. All points $p_{\text{top}}$ on $S_o$ which have a distance to the CoG higher than some threshold $t_{\text{min}CoG}^{\text{HandPoint}}$ will be treated as potential hand positions.
4. Find correspondences for all points $p_i^o$ in the frontal views and reconstruct them with respect to the world coordinate frame.

To reconstruct the potential hand positions in the world frame the correspondence problem has to be solved between the overhead and frontal views. Our approach is to solve the correspondence problem between the overhead view and every single frontal view in separate steps.

The correspondence problem between two views is less critical. For every $p_i^o$ compute the epipolar line in the frontal view and search for the closest point to it. Mark the two points as corresponding if the distance is less than some threshold $d_{\text{max}}^{\text{point2view}}$. This epipolar constraint is illustrated in Figure 5.

![Figure 5: Constraints used for hand detection between overhead and a frontal view.](image)

At the end, the algorithm comes out with the 3D coordinates of $N$ potential hand positions and passes them to the validation step 2.4.3.

### 2.4.2 Algorithm 2 - Without an overhead camera

Algorithm 2 directly uses the reconstructed points found by solving the N-view correspondences. Basically, it just passes these points without the previously detected top of the head point directly to the validation step 2.4.3.

### 2.4.3 Validation

There are two main reasons that can lead to a wrong hand position:

1. We are using wrong correspondences for the 3D reconstruction.
2. We really have a wrong hand position (for example a feet).

In both cases, a simple validation prevents from using them. The idea is to discard all points that lie outside a cylindrical volume described by two simple constraints around the main body axis:

1. The $Z$ coordinate must be within a certain range $[z_{\text{min}}^{\text{hand}}, z_{\text{max}}^{\text{hand}}]$.
2. The distance to the main body axis must be within a certain range $[d_{\text{minaxis}}^{\text{hand}}, d_{\text{maxaxis}}^{\text{hand}}]$.

Points falling outside this volume are treated as outliers and are skipped. If no point matches this validation step the whole algorithm will stop because no pointing gesture has been detected.

If more than two points have passed the validation step we use the Kalman prediction (if available), which we compute for every frame from the position detected in the previous frame, to detect the outlier(s):
1. If the Kalman predictions are available, choose the two points which are closest to these predictions.

2. If the Kalman predictions are not available, choose the two points which have the biggest distance to the main body axis.

3 GR library

3.1 Distribution files

The distribution consists out of the following files:

- Makefile
- grInterface.h
- main.cfg
- main.cpp
- GR.pdf
- doxygen/
  - index.html
  ... 
- lib/
  - libhumor.so
  - libfftw.a
  - librfftw.a
  - libouel.so
  - libvcl.so
  - libvnl_algo.so
  - libvnl.so
  - libstdc++.so.5
  - libnetlib.so
- matlab/
  - calcF.m
  - pp2f.m
- src/
  ... 

The C++ interface grInterface.h is all you need to include the pointing gesture recognition into your application. main.cpp is a small test program which creates a GR instance. It can be used to test the linkage of the library what caused the most problems so far. Use Makefile to compile and link it.

The subdirectory doxygen/ contains a HTML documentation generated by Doxygen. Use index.html as a starting point.

The subdirectory lib/ contains all libraries used for linking applications with the GR library. All these libraries have been compiled with gcc-3.2 on Linux.

The subdirectory matlab/ contains a small utility named calcF.m which computes the fundamental matrices $F'$ from the camera matrices $P'$. See section 4.1 for details.

The subdirectory src/ contains the library sources.
3.2 API

The C++ header file `grInterface.h` gives you access to the pointing gesture recognition functionality of the GR library. The methods are supposed to be self explaining, see `grInterface.h` for details.

The most important point is that the constructor is hidden. Use the global method `createGR()` which returns a pointer to a new `grInterface` instance.

The main API method is `int feed()` which takes one silhouette for a specific camera view. Since silhouettes are generated also for non-static parts of the background which have been false segmented, it is obvious that only the largest silhouette detected in the image should be passed to this method since it is most likely to one of the user.

The method returns -1 if no full set of silhouettes for the current frame is available yet. Once a silhouette for each camera is available for the current frame, the algorithm will be ran and the method return 0.

Note: a return value of 0 does not necessarily mean that a pointing gesture has been detected, it just sais that the algorithm was ran. To check if a head- and/or hand positions have been detected, use the `bool hasEyesPosition()` and `int getHandPositionsCount()` methods.

4 User’s Guide

This section gives hints to the most important questions such as “what do i need to run the pointing gesture recognition?” or “how do I tune the parameters in the configuration file?”.

4.1 Calibration Matrices

The only data input which is needed are the camera matrices $P$ and the fundamental matrices $F^j_i$ for all camera pairs.

The $P$ matrices are computed automatically by the self-calibration. The fundamental matrices can be extracted from them by using the matlab tool `calcF` (see 3.1).

Place the $P$ and $F$ matrices to some directory and set the corresponding entries in the main configuration file (see 4.3).

4.2 Choosing a well suited set of cameras

The most important point is the selection of a well-suited set of cameras for the pointing gesture recognition. Once we know where the user will probably be pointing at (in our case the projection wall(s)) we can decide about the cameras.

Figure 6 shows a camera configuration used for the pointing gesture recognition if an overhead camera is available.

First, if an overhead camera is available, it should be included in the camera set because it has a way different viewing direction than the frontal cameras and therefore improves the correspondence solving significantly.

Next, a set of cameras facing the user from front has to be chosen. There is no rule about the number of cameras, it just pointed out that the minimal number of cameras should follow the following rule: for every direction the user is pointing, the pointing arm should be seen from at least three cameras from three angles as different as possible. Example: assuming that the user will be only pointing ahead, a good set would be the overhead camera, one camera from the left or right and one camera from a 45 degrees angle. The optimal set of cameras therefore is the one that satisfies the above rule for all possible pointing angles of the user.

It is recommended to keep the set of cameras as small as possible, i.e. the minimal set of cameras satisfying above rule. Additional cameras can be added, but as they can improve the quality of the recognition as well they can cause more false correspondences and therefore false points.
Another problem are people standing in the background, for example for demo purposes. Since the whole algorithm is based on the silhouettes, it is very fragile to bad segmentation. Therefore, non-static parts of the background should always be avoided to be in the field of views of the cameras. This is not an easy task, but a trade-off between a perfect set and good segmentation can be found in most cases.

4.3 Tuning the configuration file main.cfg

This section gives detailed informations about the entries in the configuration file main.cfg.

Note: entries that are not commented in the sections below won’t be used by the pointing gesture recognition. Usually, only the testing framework needs them.

[Calibration]

MatricesF = F%i_%i.dat
MatricesP = camera%i.Pmat.cal

Set this paths to the location(s) where the $P$ and $F$ matrices are placed. Note: if the library can’t find one of the matrices for one of the cameras, it will exit (-1) immediately.

[2DClient]

MinimumSilhouetteLength = 150

Defines the minimum length of a silhouette. Silhouettes smaller than this value are ignored since they usually come from a false segmented part of the image or just provide too less information for the algorithm.

[PPP]

Cameras = 12 1 3 7 9 16
The indices of the cameras that will be used for the pointing gesture recognition.

OverheadNo = 12

The index of the overhead camera. Must be included in the Cameras list.

FirstCameraIndex = 1
LastCameraIndex = 16

Start- and end index of the camera numbers. These values are needed since the cameras for the RZ BlueC installation have a different index range \([3..18]\) as the one from Hoenggerberg \([1..16]\).

CorrThresholdPointToView = 20

Corresponds to the threshold \(d_{\text{point2view}}^{\max}\) in section 2.4.1. Should be about 20, not too small and not too big: too big results in bad correspondences, too small results in missing correspondences. Is used by the algorithm 2.4.1 which uses the overhead camera.

CorrThreshold3View = 30

Corresponds to the threshold \(t_{3\text{view}}^{\max}\) in section 2.2.1. Should be about 30, not too small and not too big: too big results in bad correspondences, too small results in missing correspondences.

[HeadTracker]

Head2EyesDistance = -0.1

Corresponds to the distance \(d_{\text{top eyes}}^{\max}\) in section 2.3 which will be subtracted from the measured \(Z\) coordinate of the top of the head point in order to get the 3D coordinates of the eyes.

MinimumHeadZ = 1.60

Corresponds to the threshold \(t_{\text{shoulders}}\) in section 2.3. It is obvious that this value depends on the height of the user.

UsePrediction = 1

Set it to 1 of you want to use the smooth Kalman prediction as output, or 0 if you want to use directly the measured point. If no prediction is available, the output will always be the measured point even if a prediction would be available.

[HandTracker]

Method = 1

Set it to 1 for the algorithm described in section 2.4.1 or 2 for the one in 2.4.2. It has pointed out that the algorithm 1 with the overhead camera is more robust in most cases. So if there is an overhead camera available, it is recommended to use it.

Hand2HeadPixelDistance = 150

Corresponds to the threshold \(t_{H_{\text{HandPoint}}^{\min}}^{\text{CoG}}\) in section 2.4.1.

MinimumHandZ = 1.00
Corresponds to the threshold $z_{\text{hand min}}$ in section 2.4.3. Depends on the user.

MinimumHandRadialDist = 0.6

Corresponds to the threshold $d_{\text{min axis hand}}$ in section 2.4.3. Depends on the user.

MaximumHandRadialDist = 1.5

Corresponds to the threshold $d_{\text{max axis hand}}$ in section 2.4.3. Depends on the user.

DetectHowManyHands = 1

Gives the number of hands to be tracked. Can be either 1 or 2 (obviously :). Because of occlusion of the hands it is more robust to track just one hand if the user is expected to always perform a pointing gesture with one hand.

MaximumDistToOldPointResetKalman = 0.3

If the new detected hand point has a distance larger than this value to the old point, the kalman filter will be reset.

MaximumDistToOldPointAsOutlier = 0.8

If the new point has a distance larger than this value to the old point, then the point will be dropped (possible outlier).

UsePrediction = 1

Set it to 1 if you want to use the smooth Kalman prediction as output, or 0 if you want to use directly the measured point. If no prediction is available, the output will always be the measured point even if a prediction would be available.

[ExtremaDetection]

FilterType = 1

Specifies the filter type:

- 1: BOX Filter (works almost everywhere)
- 2: FFT Filter (good for regular grid silhouettes as the ones of Esther)

The box filter works almost everywhere, but for regular grid silhouettes the FFT filter may work better. But the box filter seems to be a good choice for all cases and therefore is recommended.

BoxFilterLength = 7

Length in percent of the BOX filter compared to the silhouette length. More smoothing means less extremal points and therefore better correspondences. But through the smoothing the local extrema in the distance function may be shifted which, on the other hand, leads to less precise extremal points and therefore bad correspondences. 5% - 10% are good values for nice smoothing.

FFTCutoffFreq = 50
FFT cutoff frequency. The FFT has a hardcoded length of 4096. All frequencies above this value will be suppressed. The effect is the same as described for BoxFilterLength. Values between 20 to 50 seem to work best.

file0 = silhouette%i.ascii
file1 = distances%i.ascii
file2 = distancesSmooth%i.ascii
file3 = distancesPure%i.ascii
file4 = centroid%i.ascii

Filenames for intermediate results that can be written to the harddisk as ASCII files. The ASCII output can be turned on through the ExtremaDetectionFileOutput switch in the section switches. They are thought for testing purposes and therefore not suited for online use because they slow down the algorithm significantly.

[Kalman]

file = kalman.ascii

Filename for the output of the Kalman predictions for the head- and hand positions. The ASCII output can be turned on through the KalmanFileOutput switch in the section switches. This is thought for testing purposes and therefore not suited for online use because they slow down the algorithm significantly.

[switches]

Contains switches for testing file output. A switch is turned on of it’s value is 1. Any other value will turn the switch off.

ExtremaDetectionFileOutput = 0
KalmanFileOutput = 0

References


