Plane rectification in real time on an Android device
Milestone report

Jean-Baptiste Boin
Stanford University
jbboin@stanford.edu

1. Introduction

The development of reliable keypoint detectors, like the Harris point detector or the difference of gaussians, compact descriptors, as well as techniques based on scale-space theory, like SIFT [1], made it possible to do image-based search on a large scale. The latter method, which combines detector and descriptor, became very popular in many image retrieval applications. Its scale-invariance, rotation-invariance, and even its robustness to slightly different light intensities due to the use of gradients, give it many assets which justify its popularity. However, it is to be noted that the SIFT descriptor is not viewpoint invariant. Taking the picture of a same building façade from two very different viewpoints can seriously decrease the performance of the SIFT method and in that case only very few matches will be found. See figure 1 for an illustration of this phenomenon. This can be a problem if we want to recognize a building in an image taken from a very different viewpoint than the image present in the database.

Our project aims at finding a way around this limitation in the case of simple vertical planar geometry, which is an assumption that should hold for most of the man-made surfaces that surround us. As will be described later, the main idea is to find the vanishing point of the plane formed by horizontal lines. The main highlight of the project is that we want to study ways of doing our vanishing point search in real time since we want to implement it on an Android device. It will be useful to augment the data from our camera with the measurements from other sensors that a regular Android device can deliver. We will see for instance that knowing the gravity direction already partially solves our problem. Figure 2 gives a high-level overview of our project.

The ultimate goal of this project was to get a running real-time prototype on an Android device. If pointed at a vertical surface (wall, poster, painting, etc.), the device detects the orientation of the plane and corrects the view accordingly. This should be done within a few seconds.

Figure 1. Typical case of failure of SIFT due to very different viewpoints : only 15 matches are found. Credit : J.M. Morel, G. Yu , *ASIFT: A New Framework for Fully Affine Invariant Image Comparison*

2. Previous work and justification of our approach

2.1. Review of previous work

A few ways to "augment" SIFT to make it viewpoint invariant have been proposed in the litterature. The most obvious one would be to augment our database with a few additional views that would allow us to have matches from any viewpoint. Indeed, SIFT was shown to be still quite robust to moderate variations of angles (as we showed it in out

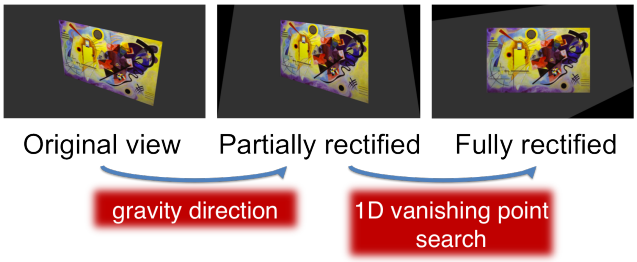

Figure 2. Overview of our project methodology. The core component lies in the 1D vanishing point search, and we propose 3 different approaches for that.

preliminary experiment, presented in the next subsection). The drawbacks are obvious :

- our database will grow substantially as the number of views per object/landmark increases
- depending of the way our data is acquired, it may be more expensive or impossible to acquire these additional poses
- the size of the database is not the only issue : more data means more matches to do if we want to compare our query image to all of our images

In some applications, this approach may not be scalable. For example, on mobile devices the computing resources and memory are usually limited, so having to perform more matches or storing a larger database can become real issues.

Another way to go around the limitation of SIFT is to augment SIFT by making it affine invariant. A notable example is ASIFT[2], which works for any kind of plane. The main feature of this method is that if we want to match two images, these images will be affinely distorted in different directions and scales properly chosen, and then the algorithm will attempt to find the pair of images with the higher number of matches. This method works well but requires extracting SIFT descriptors from many additional images, so it would not perform very well on a mobile device.

In theory, if we have only one view of the plane in our database (view that we can consider frontal), the most efficient approach would be to perspectively distort our query image so that the plane is seen from the frontal view again. After the pre-processing, the two planes should have a very similar appearance and it would be easier to match them using SIFT. This project proposes 3 different methods to solve this pre-processing rectification step for vertical planes, using the data that we can get from a mobile device.

It is important to mention that we have not find evidence of research being done in vanishing point estimation by using the gravity direction. Gravity direction has already been used in SIFT to align the descriptors with the gravity, so that they would be consistent between 2 images [3], but not for out current task yet.

2.2. Preliminary experiment validating our approach

As a preliminary experiment, we tried to see if rectifying an image actually led to improvements in terms of feature matching. To do that, we generated a virtual view of a given textured surface, similar as the images in figure 2. The camera would then be rotated to see the surface from many different azimuth angles (the parametrization of our angles will be defined later in this report) so that we would get views from different slanted angles, starting with a camera facing the plane and ending with a camera seeing it from its profile. We then ran SIFT on these images and counted the number of geometrically consistent matches between the image and the original texture. Since we know the exact position of the camera, we can also perfectly rectify the view as if we were facing the plane, as will be detailed later on. We can then run the same SIFT based experiment and compare the number of matches in both cases. The results are shown in figure 3. As you can see, the number of matches sees an important drop as soon as the optical axis of the camera has an angle of more than $30° - 50°$ with the normal of the plane. This confirms what was said before.

But it is equally interesting to see that rectifying the image considerably slows down the drop, and even at angles of $60°$, the performance drop is still very limited. This experiment shows that if we know the orientation of the plane in our query image and the orientation of our database image to compare it with, it is very interesting to rectify the query image so that we can get more significant matches, especially if this rectification can be done in almost real-time which would not add considerable overhead to our image search.

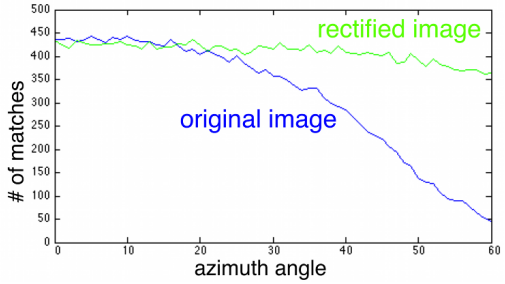

Figure 3. Result of our preliminary experiment

3. Technical part

3.1. Theoretical aspects of plane rectification, summary of our method

3.1.1 Reducing the problems to one dimension

The interesting point about using a modern mobile device is that we have access to not only the image taken by the cam-

era, but also to many different measurements given by the sensors of the device (accelerometers, gyroscopes, magnetometer). It seems that the most helpful measurements are the static values given by the accelerometers. If we consider that the device is stable while the image is captured, then the only acceleration would come from the gravity, and thus we would know its direction. We also assume that the measurements are not noisy, so the gravity direction is known exactly. Using the parametrization of figure 4, we are able to know the twist and the elevation of the camera. Only the azimuth angle remains unknown and this problem is reduced to a one-dimensional problem. This is the reason why we only consider the rectification of vertical planes, where the normal has only one degree of freedom around the (known) vertical axis.

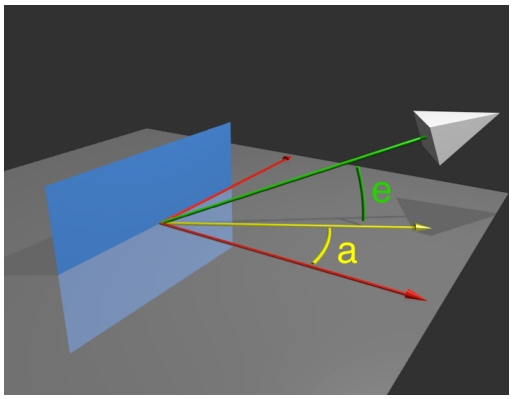

Figure 4. Parametrization of our angles. The elevation angle is the angle e, the azimuth angle is a. For clarity, the twist (rotation of the camera about its optical axis) is not represented.

One could argue that we could use the magnetometer (compass) to have an estimate of the azimuth angle. However, we cannot assume that we know the orientation of the plane at first. Also, if the accelerometer values are usually quite accurate, the values given by the magnetometer are not always as accurate, and it would be more risky to consider these values as perfect.

3.1.2 Rectifying a perspectively distorted image knowing the gravity direction

It is interesting to notice that if we consider that we know the calibration matrix, knowing the twist and elevation angles allows us to create a virtual view where the position of the camera does not change but where it is rotated so that the elevation and twist are set to 0. In other words, the view is partially rectified as the image plane becomes vertical.

Correspondances of points on a plane between two camera views a and b are related by the 3D plane to plane equation[4] :

$$^{a}p = K_a \cdot H_{ba} \cdot K_b^{-1} \cdot {}^{b}p$$

with ^{a}p and ^{b}p the correspondance pair in the image plane of camera a and camera b respectively ; K_a and K_b the calibration matrices ; and H_{ba} the homography matrix from b to a. This homography matrix can be expressed in terms of the transformation between a and b as well as the parameters of the plane :

$$H_{ba} = R - \frac{tn^T}{d}$$

where R is the rotation between a and b, t the translation between a and b, n the normal of the plane and d the distance between camera a and the plane.

In our case, t will be zero since we just want b to be a rotated version of a, and K_a and K_b will also be equal to K, the calibration matrix of our original camera, so the formulas just become :

$$^{a}p = K \cdot H_{ba} \cdot K^{-1} \cdot {}^{b}p$$

$$H_{ba} = R$$

So, once we get the different parameters of our rotation matrix, rectifying the image is straightforward since we just have to invert the formula above $^{b}p = K \cdot R^{-1} \cdot K^{-1} \cdot {}^{a}p$.

At this point, one angle is still missing in our rotation matrix (azimuth), but we can partially rectify the image as a first step. This is what we do here. We can notice that after this step, the vertical direction is completely rectified : a vertical line in world space should appear vertical in this partially rectified view, as we can check on figure 5.

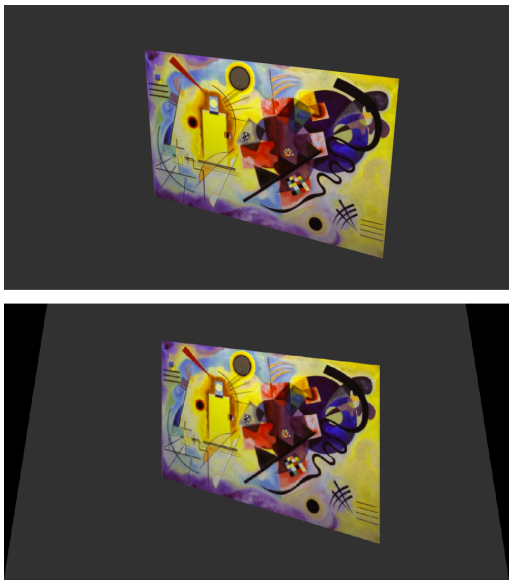

Figure 5. Original image (top) and partially rectified image (bottom), in the case of a view with a 0° twist, 20° elevation and 30° azimuth.

Now, let's try to find the position of the horizon line in this new view. We define two coordinate systems in our

world space, one associated to the plane (origin at the intersection between the optical axis of the original camera and the plane), and another one associated to the virtual camera. A top view of these coordinate systems is given in figure 6. Our partial rectification ensures that $y_c = y_p$ (the z axis is not vertical for consistency with the usual conventions on camera coordinate systems, like the one used in OpenCV). Putting aside the translation component, the camera coordinate system is in fact the rotation of the plane coordinate system about the vertical axis ; the angle of rotation is the azimuth angle we are looking for.

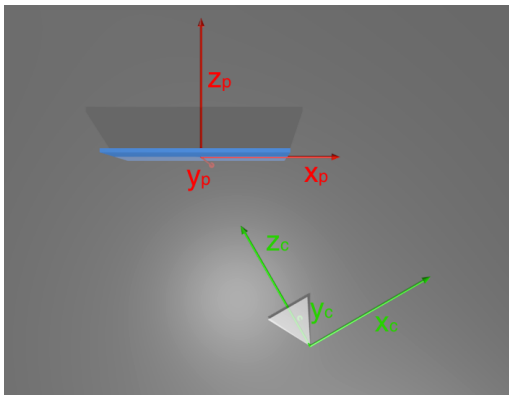

Figure 6. Coordinate systems of the plane and the camera, top view (the vectors y_p and y_c, which happen to be equal to one another, are pointing downwards).

In the camera coordinate system, a horizontal line has the form $[a,\ 0,\ b,\ 1]^T$, so the corresponding vanishing points can be parametrized by an angle : $[-\cos\theta,\ 0,\ \sin\theta,\ 0]^T$. To get the horizon line, we just need to multiply two of those using the cross product. This gives (up to a scaling factor) :

$$l_\infty = [0, 1, 0, 0]^T.$$

We can now project using the virtual camera matrix. In that coordinate system there is no translation or rotation so the horizon line is just given by

$$l_{hor} = K \cdot [0, 1, 0]^T.$$

If we translate the image coordinate system at the principal point and we make the assumption that the camera has zero skew (which is an assumption that holds in modern cameras), then K is diagonal so, up to a scaling factor, we have $l_{hor} = [0, 1, 0]^T$. In other words, the horizon line contains all the points so that $y = 0$ in the image coordinate system. In short, the horizon line is just the horizontal line going through the principal point, which makes its detection automatic once we have the sensor information.

Now, to estimate the azimuth angle a, a convenient way would be to use the horizontal lines that may be present on the plane. In that case, in the 3D camera coordinate system, the vanishing point would be $v = [-\cos a,\ 0,\ \sin a,\ 0]^T$, and so its projection in the image would be

$$p = [-f_x \cot a,\ 0, 1]^T.$$

This formula directly relates the position of the vanishing point with the orientation of our plane, which is what we are looking for.

To summarize, our rectifying approach consists in partially rectifying the image to make the horizon line very easy to parametrize. Then, we can run a one-dimensional search on this line to find the vanishing point of our plane, and to finish, we can find our final angle a using the formula above. We can then fully rectify our image using the 3D plane to plane equation.

3.2. Three ways to find the vanishing point

We developed three different methods to find the vanishing point of a surface. We did not develop them at the same level of polish but they all try to meet different goals.

3.2.1 The line-based approach

The first approach we tried is probably the most logical one. On a surface where some horizontal lines are dominant, the basic approach would be to detect the dominant lines, and see where they intersect. This is what we started to implement in Matlab. After the partial rectification, that ensures that the vertical lines are rectified and that the horizon line is known, we first run a Canny edge detector on the image, and then we detect the most prominent lines using the Hough transform. After populating the accumulator array by processing all the edge pixels, we look at the highest peaks, which correspond to the dominant lines in the image. We then go back to our original image and segment these lines into segments. At this point, we know the main segments in our original image, as well as the horizon line.

But if we expect that some of these lines will indeed intersect on the horizon line, we cannot make the assumption that all of them do, since the plane will not only have horizon lines in world coordinates. So we are facing with a problem of having many outliers. One way to deal with that is to have a running point on that line and to estimate at each position the likelihood that this point is the vanishing point. For each point, we loop through all the lines and add a high score if the point is almost aligned with it, while the score will be low if it is not. Figure 7 defines the angle parameter that we used for this approach. M_2, which has lower angles than M_1, will thus receive a higher score than M_1 and will be considered more likely to be a vanishing point.

Another refinement that we added was to consider the length of the line. What we noticed when experimenting with the above algorithm was that it was very sensitive to noise, since all the detected lines are used in the same way.

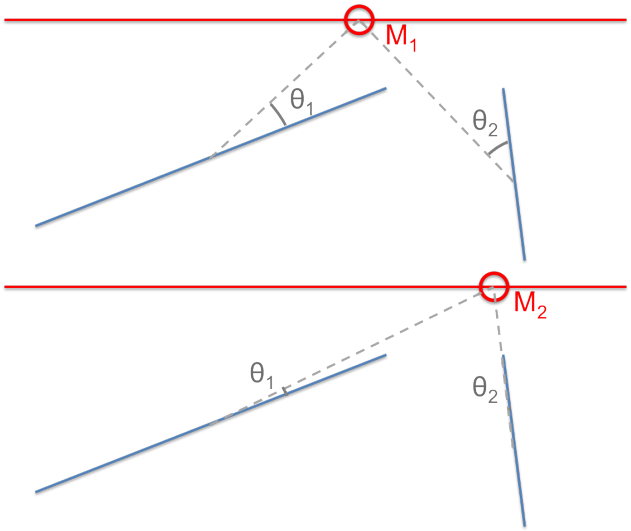

Figure 7. Exemple of a configuration where the red line is the horizon line, the two blue segments are the ones detected on the image, and M is the running point, shown in two different positions.

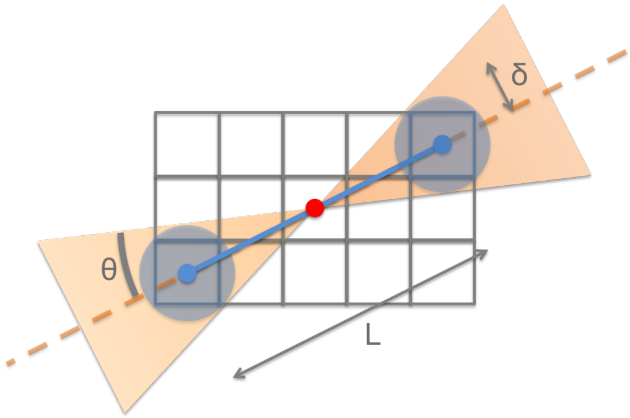

Figure 8. This diagram shows the uncertainty in orientation depending on the uncertainty on the endpoints of a line. If this uncertainty in position is in the order of $\delta = 1$ pixel, then the uncertainty in orientation is given by $\theta = \arctan(2\delta/L) \approx 2\delta/L$, which justifies our choice of using a gaussian with standard deviation proportional to $1/L$.

But on real data, we would usually have clear long horizontal lines (edge between a wall and the ground for example), and many other unreliable smaller lines, that could be in many different orientations. If a line is very short, then pixel quantization makes it easy to badly estimate its orientation. Figure 8 gives a good idea of this phenomenon. In that case, when estimating the score, we should take into account the fact that values of θ not that close to 0 should not be penalized too much. However, the orientation given by a long lines is much more reliable, so we would expect the score to be very high when θ is close to 0, and to drop very fast as θ increases. A good way to model that was by using gaussian distributions with a standard deviation inversely proportional to the length of the line. The distribution corresponding to a long line will be very peaked around $\theta = 0$ while a shorter line will have a more flat distribution.

A last problem that we face is that the horizon line is infinite, so we need to sample it in a smart way. Given the formulas we found above, it made sense to sample it for all the

$$x_k = f_x \tan(\theta_k)$$

where the angles θ_k would be linearly spaced in the interval $(\pi/2, \pi/2)$. Finally, after we ran this algorithm for every point of the horizon line that we sampled, we can look at the sum of the scores for each point and the higher one will give the vanishing point.

As we will see in the results section, this method has the advantage of being quite accurate. But as the reader may have noticed, many operations are involved, and our dimensionality reduction that we got from using the gravity direction may not have been exploited as much as it could. In fact, running a full Hough transform on our edge image is expensive and it is the bottleneck of this method. This led us to consider if there would be a way to do without this Hough transform. This leads to the second method, the point-based method.

3.2.2 The point-based approach

Running a full Hough transform allowed us to select the edge pixels that were on lines and to discard the others. This is something that is desired if the main goal is accuracy, but maybe not if it is speed, since it involves a computation intensive step. This second approach will consider all the pixels directly, without running a search for lines first.

In the previous approach, we used the orientation that we found when we looked for lines in the image. But if a point is on an edge, then its gradient should be orthogonal to the edge direction, which gives us a rough estimate of the orientation of the edge. And this estimate comes at almost no cost since computing the gradient is actually part of the edge detection. Using a scoring scheme similar to the previous method, we would expect that the contributions of pixels on an edge would add up to increase the likelihood as wanted.

This time, we used a distribution for the scores that was only dependent on the angle θ between the local edge orientation and the line joining the running point on the line of horizon and the pixel considered. Since this method is much more noisy (the gradient orientation is not that accurate), we chose a very flat distribution for our scores. The upsides of this method is that we ge a higher speed, but it comes at the cost of accuracy as we will see in the results.

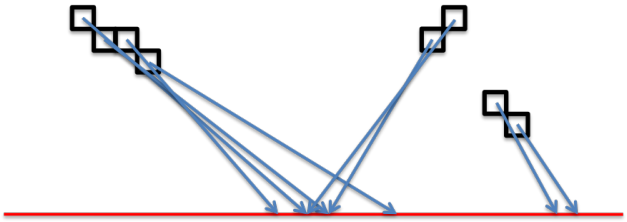

Figure 9. Every edge pixel contributes on the line

3.2.3 The texture-based approach

This last approach is the least polished one, but it still gave some interesting results. In this approach, we try to ask the question of what would happen if we do not have a plane with strong horizontal lines but instead some kind of texture, like an irregular brick wall or a wallpaper. If we partially rectify our image, we know that the vertical direction is properly rectified and that there is no perspective distortion on that column for the points of the plane. If the plane has a texture with a typical scale, it means that this scale will be preserved within a column. However, the interesting point is how different columns compare : indeed, because of perspective distortion, the typical scale should decrease as we get closer to the vanishing point, and this decrease is linear, as we can see on figure 9.

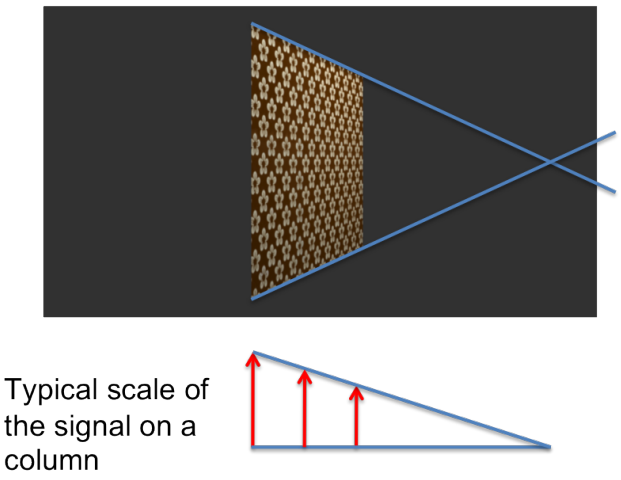

Figure 10. The scale of the pattern goes down until it reaches 0 at the vanishing point.

Using this observation, we could be able to detect the position of a vanishing point by just looking at the typical scales of all the columns and trying to fit a line through them. The point when this line reaches zero would give the position of the vanishing point on the horizon line.

More formally, we used wavelets to get the typical scales within a column, because they are very adapted to a multi-scale analysis. We run this algorithm for each column, which gives us a representation of the signal in the scale space (cf figure 11). We then select all the local maxima, which is motivated by the fact that within a column, the maxima should be located at more or less the same scales. Then we count the number of maxima that we get for each scale, which gives us a histogram.

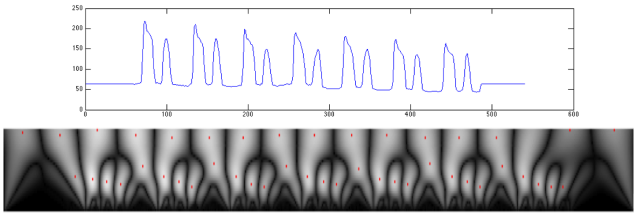

Figure 11. Signal corresponding to a column of figure 10 (top). Representation in scale space, the scale being the y-axis. The maxima are indicated in red. Their projection on the scale axis will give us a histogram that will be our feature vector.

After doing that for all the columns, we get a 2D histogram like the one in figure 12. This figure shows very clearly that the maxima are aligned along lines that will meet at 0, where each line corresponds to a different scale of the pattern of the wallpaper. The position of this point will give the position of the vanishing point. If this example is too perfect to be very realistic, we could still observe a decreasing trends when we used the same layout with different textures. Once we get this histogram, we found two ways that did a relatively good job at finding the right position for the vanishing point.

The first one is an optimization function using a RANSAC loop : we draw a line between two points and we look at the distance between the line and the points. We then scale these distances according to an optimization function (we found that using a function that is quadratic close to zero and then constant gave good results, since it doesn't penalize outliers too much), and add them. We then select the line with the lowest score. The second method was to use a Hough transform, and to fit the line that would correspond to the maximum peak. Once again, the use of the Hough transform is justified by the fact that our way of obtaining the diagram is very noisy and we have many outliers.

These methods did not always work, but using one or the other we could usually get a good estimation of the vanishing point. This approach still has many flaws, and polishing it could be an interesting research direction. It is obviously not practical for a real time approach since running a wavelet transform on all columns require too much time.

Still, the fact that we could get a good estimation of the vanishing points in some cases where the texture was irregular was quite encouraging. However, we did not have time to identify what made our algorithm fail in many cases. With more time, we would have liked to try a 2D wavelet approach and compare it with our current results.

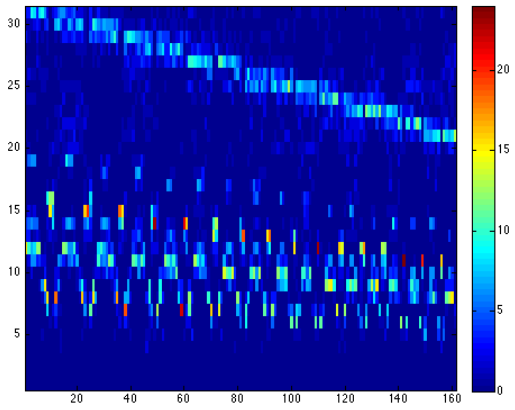

Figure 12. 2D histogram of our scale maxima. We can see clearly different linear trends.

4. Results

In this section, we will study the different results that we obtained and compare our algorithms.

4.1. Database

We created two different datasets for our purpose. The first one was computer generated, using Blender, which allowed us to have ground truth images, where the camera parameters are known exactly, as well as the different angles. A few samples of this dataset are given in figure 13.

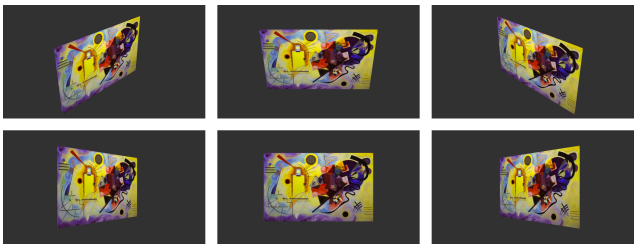

Figure 13. Different views of the same plane seen from different angles : e = 30° for the first line, e = 0° for the second; a = -45° for the first column, a = 0° for the second and a = 45° for the third.

The second dataset was collected by taking pictures of buildings of Stanford (outdoors and indoors scenes) to try our algorithm on actual data. Unfortunately, this real data was not labelled since it was not easy to know the azimuth angle between the camera and the plane normal without actually measuring it, which we did not do. Still, we could try our algorithm on this data and see how well it performed by visually judging if the planes were rectified.

4.2. Results of our three approaches

The line-based approach can be considered as the standard one, since it is undoubtedly the most reliable of the three. It is also the one that we could test the most intensively. Here are the results that we got from our synthetic dataset. The results are given in figure 14. Each rectangle corresponds to an outcome of our algorithm (the images being sampled according to their azimuth and elevation angles). We then plotted the difference between the azimuth we found and the real azimuth.

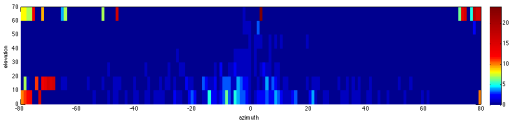

Figure 14. Validation of our line-based algorithm on our synthetic database.

We can see that our results are very encouraging. Our algorithm is accurate within a couple of degrees almost everytime, until it breaks when the view is too slanted, at an azimuth of around 80°.

When tested on our second dataset (real data), we still got very good rectification results, as we can see in figure 15. None of the images that we tried gave totally wrong rectification results.

Figure 15. A few results on real data. The left images are the original ones, and the right ones are the rectified ones. We superimposed actual horizontal and vertical lines in red to compare with a ground truth.

However, we noticed a few small discrepancies, as we show in figure 16. This figure shows a partially rectified image, with the horizon line in blue and in red the vanishing point given by our algorithm. In red are actual vertical lines and in green world space horizontal lines. The reason why our vanishing point is not exactly where it should be is because one of our basic assumptions does not hold in that case. We can notice that after the partial rectification, the vertical lines are not completely vertical, and the horizon line does not go through the real vanishing point. This is in fact an example where the photo was probably taken while

moving, and the gravity direction given by our sensors is slightly off. But for most images, this was not a problem, and keeping still while taking the photo was enough to get reliable accelerometer values.

Figure 16. Example of a result slightly off because of the inaccuracies of the sensors.

One last remark that we can make about this algorithm is that it could probably be extended to a search for multiple vanishing points, using the photo of a room for example. Figure 17 gives a good example of this. In this photo where there are two dominant plane orientations, our algorithm could find two peaks, one corresponding to a plane with an azimuth angle close to zero (a plane facing us), and another peak is found for a plane on our left, which is exactly what this image is.

We did not run an extensive testing sequence with the point-based algorithm, but comparing the results between this approach and the line-based approach could confirm our intuition on its lower accuracy.

What we see in figure 18 is striking : if both results manage to get the right azimuth (30°), the point-based approach has a much higher background noise than the line-based approach. This is due to the fact that many edge pixels that do not belong to any line will add their contribution, which leads to this background noise. Also, since we do not use any information about the length of lines, we cannot get peaked distributions like in the line-based approach. These observations show that we cannot expect to get as good results from this method in real world applications, but we can perhaps get results that are good enough given that it is much faster to compute.

Finally, we already stated that our last approach, the texture-based approach, gave unreliable results, but this is due to the difficulty of the task. To illustrate this further, we show in figure 19 different examples of textures with their 2D histogram of scales, one of them being processed successfully by our algorithm, while the second one is not.

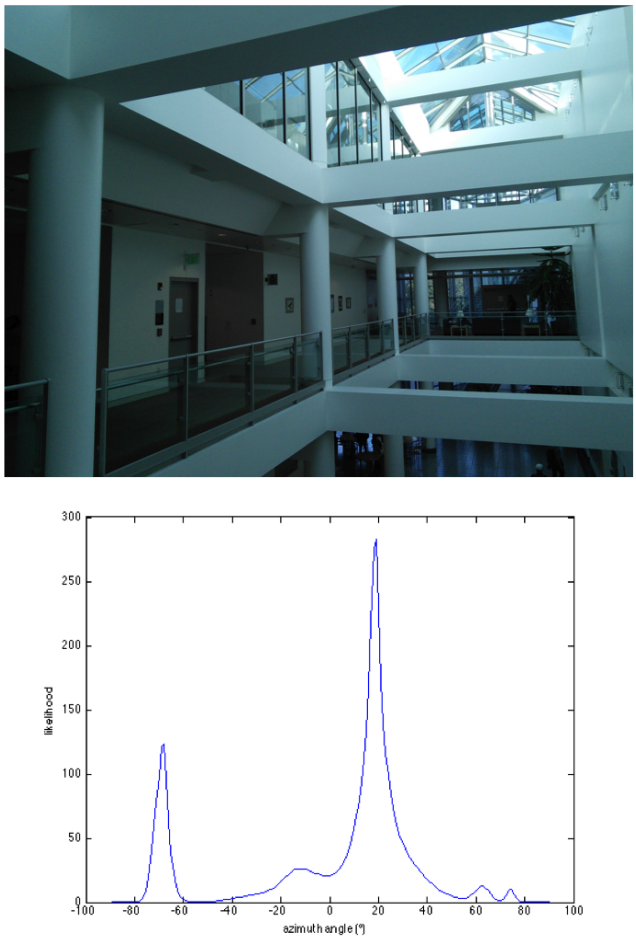

Figure 17. Original photo with 2 distinct vanishing points (top). Likelihood of the azimuth angle (bottom).

Visually, we can see that the second histogram is much more challenging and even the human eye would have trouble fitting a line through it. This is due to the fact that the pattern is less repetitive compared to the other one, so there are many multiple concurrent scales. Still, it can be considered as a success to manage to process more or less regular textures like the first one since the two other approaches discussed above would fail to find a correct vanishing point.

4.3. Implementation on Android

The highlight of this project was not only to develop realistic approaches to find the vanishing point of a plane in real time but part of the interest was also to implement them. We could implement the line-based method and the point-based method on a tablet (Nexus 7), using the OpenCV library. We will not develop the application in itself much here since it was already showcased during the poster session.

But to show the promising aspects of the point-based approach, we will give our performance results in terms of speed. When our application was in multi-threading mode,

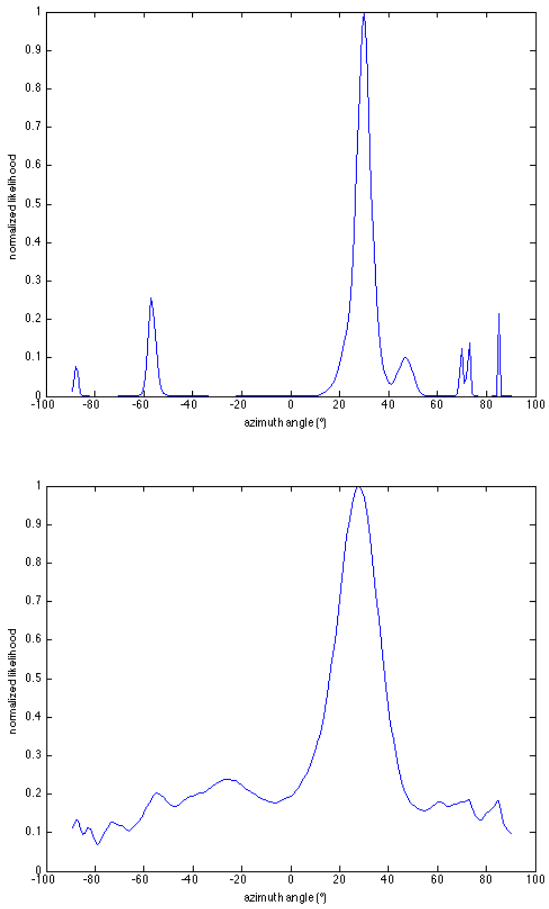

Figure 18. Normalized likelihood obtained by both algorithms when applied to the top image of figure 5. The top result was obtained with the line-based approach ; the bottom one with the point-based approach.

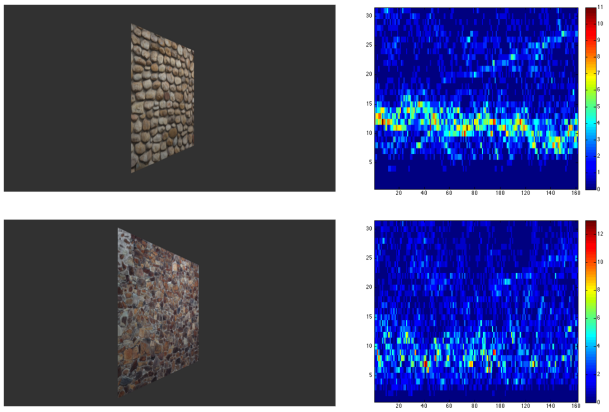

Figure 19. Exemple of textures with their associated algorithms. The first one was processed accurately while the second one gave a wrong vanishing point position.

the tablet was doing 2 things.

- The main graphic thread was taking a frame, the last computed value of the azimuth angle as well as the sensor data to completely rectify the frame
- Meanwhile, another thread would take a downsampled version of the frame (dimensions divided by 8 compared to the full resolution frame) and run the slower algorithm (line-based or point-based depending of the mode) that would compute the azimuth angle.

The performances we got when running this mode were the following :

- The full rectification of the high resolution frame took about 5 frames (with a framerate of around 30 frames per second)
- The line-based approach took around 30 frames to compute a new version of the azimuth angle
- The point-based approach took around 8 frames to compute a new version of the azimuth angle.

In the end, if we noticed that the point-based approach was, as expected, less accurate than the line-based approach, we could make it run almost as fast as the thread that ran the rectification, which means that our real-time requirement is nicely met.

5. Conclusion and future developments

It was interesting to develop three different approaches to solve a same problem. Each of those has its pros and cons. The line-based approach provides a high accuracy but takes a longer time to compute ; the point-based approach is much faster but less accurate ; the texture-based approach works in unexpected cases as soon as there is some structure in the texture of the plane but it can be difficult to get good results when the texture becomes too irregular.

For our original real-time problem however, it seems like we have developed an interesting method, the point-based approach, that can be very efficient while still giving relatively good results. A bit of additional work would be necessary to optimize it even further. For example, we could probably get rid of the first step, the partial rectification of the image. Indeed, we do not need to have a horizontal horizon line to run our algorithm, and this would decrease the computation time even further. Another direction that we would like to look at is the time coherence between consecutive frames. If we could run our algorithm fast enough, then we could expect two consecutive processed frames to give close results in terms of azimuth, especially if the accelerometers have not measured high accelerations meanwhile. Using this prior knowledge, we could temporally smooth our results to get rid of obvious bad results.

6. Thanks

I warmly thank Roland Angst as well as my advisor Bernd Girod for their support and very relevant suggestions through the project.

References

[1] D. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, 2004.

[2] J.M. Morel, G. Yu, "ASIFT: A New Framework for Fully Ane Invariant Image Comparison," *SIAM Journal on Imaging Sciences*, 2009.

[3] D. Kurs, S. Benhimane, "Inertial sensor-aligned visual feature descriptors", *CVPR*, 2011.

[4] R. Hartley, A. Zisserman, "Multiple View Geometry in Computer Vision", Second edition, *Cambridge University Press*, 2004.