Minimisation Tree from a set of RGB-D images

ALMEIDA. João* 90119, JANEIRO. João Maria† 90105, VIEGAS. Guilherme‡
90090 February 21, 2021

***Abstract*—3D scene reconstruction is an important technique in the computer vision field. Inspired by the structure from motion systems, we propose a tree that reduces the cumulative error of the rigid transformation to the world reference image, using a set of images which are gained by a commonly used camera. There are many key techniques in 3D reconstruction from image sequences, including feature matching, RANSAC, rigid transformation, projective reconstruction, etc.**

The system tracks all the images and tries to associate to another image that is already known to be connected to the world reference. The algorithm keeps checking all the neighbours with an incremental distance and evaluates the percentage of inliers detected, therefore it allows to connect non sequential images if the rigid transformation is valid. This method constructs an almost optimal tree must faster than comparing every image to all the other data set images.

The effectiveness of our algorithms is evaluated in the experiments with real image sequences.

***Keywords*—** feature matching, RANSAC, Procrustes, essential matrix estimation, projective reconstruction.

I. Introduction

The 3D scene reconstruction from multiple view images is an increasingly popular topic, which can be applied to street view mapping, building construction, gaming and even tourism, etc. The goal for this project is to develop code for 3D localization, specifically, given an RGB and depth image dataset, compute rigid transformations from each camera to a given world coordinate, in this case, set to the coordinates of the first camera frame. The production of this datasets is done via a *kinect* sensor, which has an RGB camera and an InfraRed sensor for the depth image. Because the goal is to compute Rotation and Translation matrixes from each camera to the first one, the way to do this is via building a graph where ideally each node has a rigid transformation matrix from itself to the previous frame, and the rigid transformation from itself to the first camera frame is simply the composition of all rigid transformations behind itself, so, for example, from camera 3 to camera 1 we have the composition of camera 3 to 2 and then 2 to 1. Because the camera positions may differ from the idealized or some specific camera perspective may be more well aproximated to the first camera than the previous camera, in those specific cases it's computed a direct rigid transformation to that first frame, so for example computing a transformation from camera 5 to 1 could be bettter aproximated by computing the rigid transformation from 5 to 4 and then from 4 directly to 1 instead of composing 4 to 3, 3 to 2 and the 2 to 1. Will be presented an explanation of the methodology used, as well as some more relevant results. Some considerations will also be made about the results obtained as well as possible improvements and future work.

The assignment of the project is available here and the paper here.

II. State of the art

This implementation of the tree is not optimal since the algorithm does not take in account the number of the intermediate transformations to the world reference.

The next step is to recreate a 3D scene, and for the sake of avoiding duplicate 3D points, the system reconstructs the 2D point only when one children shifts out of the boundary of a camera.

The final step would be taking the noise and moving objects into account, which could be suppressed when transforming multiples images to one frame and applying a median filter. This filter not only would remove an object that is not there all the time but also some noise caused by any transformation.

III. Proposed solution

Through the sequences of RGB and Depth images provided it's necessary to:

- Find a correspondence between point clouds, for each pair of images
- Obtain the transformation between the pairs of 3D points, that is, obtain a rotation and translation matrix that represents it
- Obtain the transformations of the various images in relation to the first image, with the smaller associated error possible

So we can divide this problem in some smaller problems, that, put together, solve the main problem.

1) Given an RGB and Depth image, relate those with the help of the respective intrinsic paremeters
2) Compute keypoints from both RGB images, so that it's possible to relate both of them
3) Using Ransac method, divide the previous obtained keypoints into inliers and outliers
4) Solve the Procrustes Problem with the previously obtained inliers, obtaining the Rotation and Translation matrix that best align both point clouds
5) To know which rigid transformation matrixes to be used, it's needed to store an associated error to that rigid transformation

IV. Data model

An RGB-D camera (e.g. Kinect) can give combinations of a RGB images and its corresponding depth images, since it has the distance in each pixel between the image plane and the corresponding object in the RGB image. Therefore with the camera intrinsic it is possible to represent each pixel in the image plane and combining the depth value with the corresponding image plane pixel, recreating the colored object in real world coordinates.

A. Camera model

A camera works by projecting word 3D points into an image plane (2D), but the inverse is impossible unless we have the Z coordinate, or depth, knowledge of each point projected into the image plane. So, considering a normalised camera(scale factor = 1), having a point P = (X, Y, Z) into the camera plane (u, v), the perspective projection is given by (1)

$$u = \frac{X}{Z}, v = \frac{Y}{Z} \tag{1}$$

Bearing in mind the intrinsic camera parameters, it can be written as a 2D transformation (2), where x and y are in meters, x' and y' are in pixels, f_i the respective focal distances, c_i the respective origin point in pixels and s_i the respective scale factors in pixel/m

$$u' = f_{s_u} u + c_u v' = f_{s_v} v + c_v \tag{2}$$

Obtaining the system in (3)

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \begin{bmatrix} f_{s_u} & 0 & c_u \\ 0 & f_{s_v} & c_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \Leftrightarrow x' = [K]x \tag{3}$$

All the intrinsic parameters for this projects are given as input.

Considering now the extrinsic parameters it can be written as a 3D transformation by

$$X = [R]X' + [T] \tag{4}$$

Which combining both intrinsic and extrinsic gives rise to (5)

$$\lambda x' = [K][RT]X' \tag{5}$$

Which, having the Z coordinate data it's now possible to retrieve the 3D coordinates from each pixel, generating the respective image pointcloud, aligning both RGB and Depth camera planes.

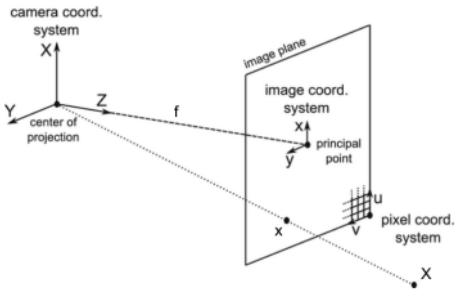
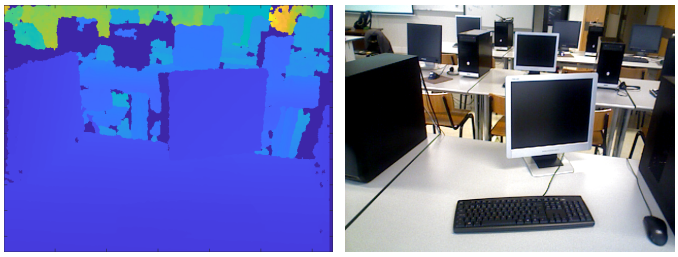


Fig. 1. Camera Model

B. PointCloud generation

In the Figure 2 is represented a RGB-D image. Analysing the depth image, the intensity of the color represents how far the real object is from the image plane. The RGB image corroborates with the last statement in the way that the shade of blue through the computers is getting bright, the wall is already green and finally the space outside the classroom is yellow.



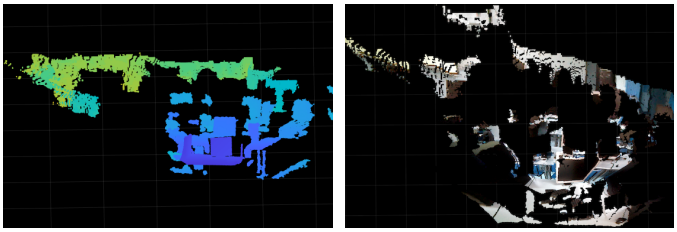
(a) Depth image (b) RGB image

Fig. 2. RGB-D image from Kinect.

In order to relate both images, first of all it is necessary to transform the depth information from the image plane in the Depth camera coordinate, which is called a point cloud and is represented in the Figure 3a. Bearing in mind that the image is represented in projective coordinates and normalised, the process to generate the point cloud is to multiply those coordinate in each pixel for the corresponding depth information.

Secondly the point cloud is rotated and translated to the RGB camera coordinates and one interesting point is to recreate a colored point cloud, that match the RGB information with the transformed point cloud in each pixel, and can be seen in the Figure3b.

1) Over computing Point Clouds: In order to not compute the point cloud of the same RGB-D image multiple times, whenever the program detects a repeated instance, it simply loads from a pickle file.



(a) Depth Point Cloud (b) RGB Point Cloud

Fig. 3. Point Clouds from different each camera view.

V. Subproblems

A. Keypoint Detection

For relating two pointclouds it's crucial to map points from both RGB images. That's done by finding keypoints, usually associated to changes on the gradient, which do not change when the image translates or even rotates. With SIFT, an algorithm for detecting keypoints for a given image it's possible to relate both RGB images. It computes the keypoints for each image by getting maximum values in the gradient of each image pixel, which correlates to changes in color or light, and so that the keypoints won't be affected by some rotation, it also assigns an orientation to each keypoint by calculating the gradient magnitude and direction in the neighbourhood region.

This maximum value is computed by getting different Gaussian planes using a window for that Gaussian (engineers love Gaussians, every time we see them, we get happy) that varies depending on a scale factor, so that now the local maxima computed depend not only on the position of the region to be computed but also on a scale factor, so that now the pixels are compared not only to its 8 plane neighbours but also to the *upper* and *lower* planes, meaning if a local maxima is found, it's a local maxima for that certain scale. With this, it's ensured that those local maxima don't change with some image rotation and/or difference in scale.

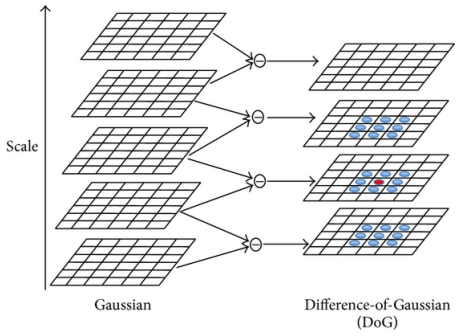


Fig. 4. Difference of gaussians

Now keypoints between both images are matched by identifying their nearest neighbours. In some cases, the second closest-math may be very near to the first, due to noise or other reasons, and in that case, ratio of closest-distance to second-closest distance is taken, rejecting if it's greater than 0.8. With this SIFT is capable of eliminating around 90% of false matches while discarding only 5% correct matches [1]. With this we now have a list of keypoints for both images to compare.

B. RANSAC

Random Sample Consensus (RANSAC) is an iterative algorithm, that receives a set of points(in this case, the keypoints previously computed) and separates them as inliers and outliers. To implement it, the next described steps were used:

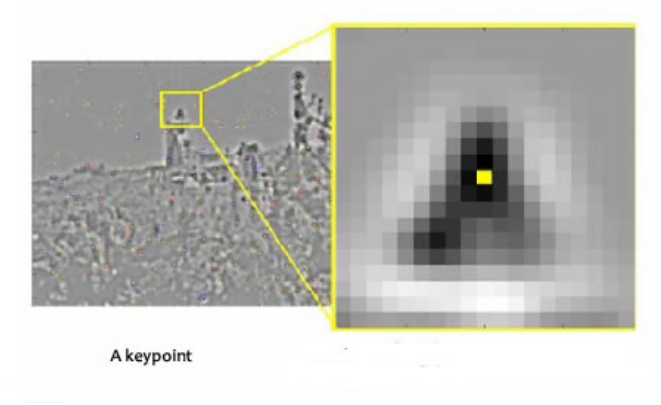


Fig. 5. Keypoint

1) Choose **4** random points from the list of keypoints received as input. The number of random points chosen is important because it's needed to choose the minimum number of points for creating the model, more points used would mean generating a model that could best aproximate some of the outliers, less points would not be matemathically correct because using a system of equations for creating the model would mean 12 parameters to discover, so the minimum number to use is four points each with three coordinates generating 12 equations to solve.
2) Apply the Procrustes problem, described in the next subchapter, to the four chosen points, generating a Rotation and Translation matrix
3) With the previously computed model, an error inherent to the model is calculaded by applying to all the keypoints, the rigid transformation computed, and check the proximity of the generated 3D point to the reference frame pointcloud, based on a *frobenius norm*. If the computed error for the generated point is lower than the defined threshold, specifically 0.2 (20cm) the number on inliers is incremented
4) Now, a new point cloud with the rigid transformation previously calculated is generated, and with this a fine tuning is tried solving a new procrustes problem between the pointcloud computed and the objective pointcloud, and the new number of inliers is checked. If the number of "tuned inliers" is greater the the inliers computed before, the compound matrices are generated
   $R = R_{tuning} \cdot R_{started}$
   $T = T_{tuning} + T_{started}$
5) The steps in 1, 2, 3, and 4 are repeated for a number
6) If the number of inliers outpasses a certain specified value, the loop is broken before reaching the maximum number of iterations, because it's ensured that it's a more than reasonable rigid transformation to have, with this, program speed is greatly improved
7) If, in the end, the maximum number of loop iterations is reached, the Rotation and Translation matrixes used are the best one found in all loop iterations

1) Random points: Choose **4** random points from the list of keypoints received as input. The number of random points chosen is important because it's needed to choose the minimum number of points for creating the model, more points used would mean generating a model that could best aproximate some of the outliers, less points would not be matemathically correct because using a system of equations for creating the model would mean 12 parameters to discover, so the minimum number to use is four points each with three coordinates generating 12 equations to solve.

C. The Procrustes Problem

For obtaining optimal rotations and translations between pointclouds, with the goal of aligning both of them, a Procrustes Problem is solved. First, it's found the centroids for both pointclouds, by averaging the respective points. So considering as A one pointcloud and B the other one, it's performed a translation (6) for centering both pointclouds to the same point.

$$\begin{cases} \overline{A} = \frac{1}{N}\sum_{n=1}^{N} A_n \\ \overline{B} = \frac{1}{N}\sum_{n=1}^{N} B_n \end{cases} \tag{6}$$

So a covariance matrix, **H**, is generated, by having $\mathbf{S} = B_c A_c^T$. Now, *Single Value Decomposition*, or SVD, is applied, to find the best rotation matrix. First it is needed to find [U, S, V] = SVD(H), where $\mathbf{H} = \mathbf{USV}^T$, U is a matrix where the columns are the left singular vectors of $\mathbf{HH}^T$, S a diagonal matrix with the singular values of H and same dimension of $\mathbf{H}^T\mathbf{H}$ and $\mathbf{V}^T$ is a matrix where the rows are the right singular vectors of **H**. **U** and **V** are orthogonal matrices and $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$. With this, it can now be computed the best rotation matrix (7)

$$R = U \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & det(UV^T) \end{bmatrix} V^T \tag{7}$$

and the translation vector (8)

$$T = \overline{A} - R\overline{B} \tag{8}$$

It's important to note that the sign of $det(UV^T)$ impacts the rotation matrix, with a minus sign meaning a reflexion on the rotation matrix, which is not what it's intended, so it's needed to ensure that this determinant evaluates to 1.

D. Graph Generation

Because computing all rigid transformations from and to all the images and then search for the best rigid transformation for each of them is computationally infeasible, and because, even if idealizing a completely continuous dataset, computing only a rigid transformation between pairs of images would generate a propagation in the error, lets say, e.g. image 4 to reach image 0 had to do 4-¿3-¿2-¿1-¿0, this would carry an error that would increase with each rigid transformation done, the bigger the chain, the more error it would generate. To solve this there was the need to create an abstract way of storing the best rigid transformations obtained, all this decreasing the cost of computation and ensuring that all images will have a good connection to the reference image (image 0).

For this, it was built a tree with each node representing an image, to store their parent (id of the node parent node), a rotation and translation matrix to transform into the respective parent. In this case, only images with an *acceptable* rigid transformation to their parent, and by *acceptable*, it means having generated more than 50% of inliers for that transformation and over 1% of points in both images being compared are keypoints are added. This ensures they are connected to the reference image, because its parent is connected to the reference image. Thereafter, with the built tree, it's now easy to recursively iterate over each graph node and make the respective matrix composition for each image rigid transformation, by walking through that tree until the reference image is found multiplying the rotation matrices on the *walk* and summing the translation vectors.

Thus, even if some tree is relatively big, the associated error will always be smaller than having only a continuous and big chain.

To start off the building of the tree, first all elements are added to a stack, ordered by the order they appear in the dataset (the order is not important). Then the nodes in the stack are visited one by one, each one tries to compute a rigid transformation with the reference image (image 0) to find a matrix R and T, that will do the transformation between each node and the reference, if the percentage of inliers found is above 50%, we include it in the graph having the reference

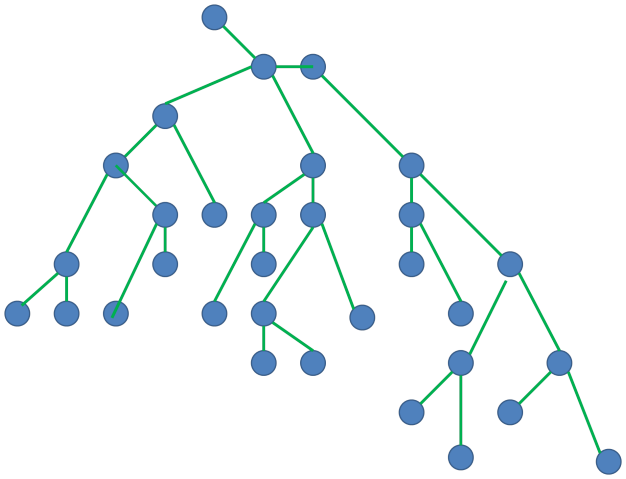


Fig. 6. Graph Tree

image as parent, otherwise we include it in a list of visited nodes not yet in the graph, where we also set its parent as the reference. This last list is called *best_unfound* because it stores the best R,T found up to now for each node not yet in the graph. This way, in the end, even if there are images left in *best_unfound* with rigid transformations not considered ideal, in the end they are added to the graph.

Next, it starts to iterate over the nodes left in the stack (the stack is now ordered by percentage of inliers in *best_unfound*), computing rigid transformations between the nodes distancing a variable that increments every time all nodes in the stack are visited, so if node 3 is inside *best_unfound* it computes rigid transformations for node 2 and node 4, because it's assumed that the datasets are relatively continuous, and if any of these transformations are considered nice transformations and the node it's comparing is already inside the graph then it can be added to the graph, otherwise it stays inside *best_unfound* but updates its node parent and rotation and translation matrices if a better than already stored transformation was found. It does that while the stack has nodes inside or until the maximum stepped distance is computed, for example for a 10 images dataset it would be a maximum step of 9, for increasingly bigger steps, so thereafter if node 3 is still inside *best_unfound* after using distance=1 then it computes rigid transformations for nodes 1 and 5 (distance 2), so on and so forth. In the end, if *best_unfound* still has nodes inside, maybe because some closed sub-tree inside *best_unfound* was created, they are deconstructed and appended to the graph even if the transformation is not the exact best.

VI. Experiments

For the office dataset, after a program run, it can now be plotted new pointclouds based on the generated Rotation matrix and Translation vector, for example for images 2 and 5 seen in Figures 7a) and 7b), showing as well the reference image used in Figure 7 c).

In Figure 8 it's showed the generated pointclouds for image 2 and 5 as well as the reference pointcloud, all overlapped to show a relatively good superimposition of the pointclouds, meaning the calculated rigid transformations were good.

To note that when trying to display the coloured pointcloud it's to hard to check for the overlapping quality and it's noticeable some strange occurrences and some increased noise on images with too few matching keypoints.

For the *office* dataset it is showed in Figures 9a) and 9b), for dataset images 10 and 23 respectively, the generated image applying the respective rigid transformation to be seen from the perspective of the camera that generates the reference image seen in Figure 7c).

In Figure 9 c) it shows the generated transformation from image 12 of the *office* dataset. It's relevant this plot because, as showed

(a) Image 2 (b) Image 5

(c) Reference

Fig. 7. Input images

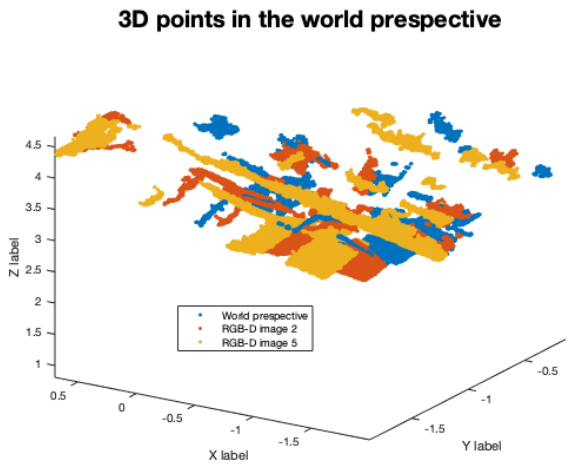


Fig. 8. Overlapped Pointclouds - Reference(blue), 2(orange) and 5(yellow)

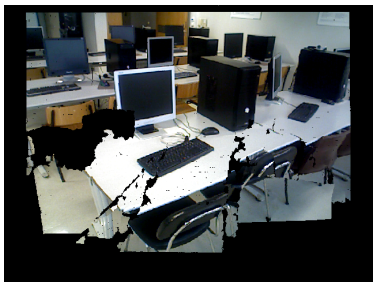
in the generated graph, in Figure 10, this image is far away from the reference image, so it would be expected some large image degradation due to the compound error that propagates, but that do not occur, which means good transformations are being made for this specific dataset.

VII. Discussion

The proposed solution offers a good way to find the best relations between a given dataset and a reference image (in this project the first image was taken as reference). The solution doesn't offer the best worst case complexity, being $O(N^3 * RANSAC)$ - N being the number of images - but the medium case is rather good. We could do an all to all approach and then apply a shortest path algorithm such as $A*$ or *Dijkstra*'s, giving a worst case of $O(N^2 * RANSAC + V + ElogV)$ - N being the number of images, V and E the vertices and edges of the graph, respectively - but in the medium case that would not offer a better time than the proposed solution. The knowledge that the dataset is continuous is leveraged and that's why this proposed solution is possible, as you can see in Figure 11. The points verify

(a) Image 10

(b) Image 23



(c) Image 12

Fig. 9. Office dataset examples

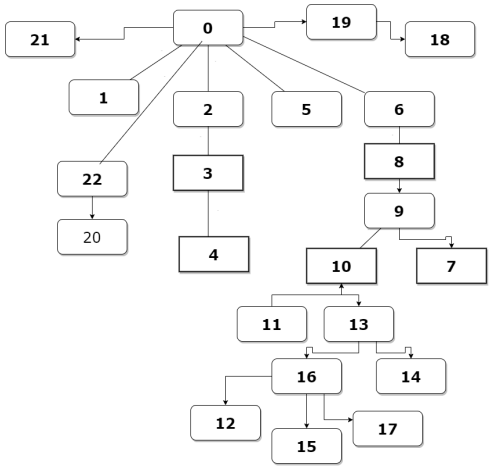


Fig. 10. Graph generated for office dataset

the best relations they have based on proximity, only going farther if needed (which is not very common to expand a lot).

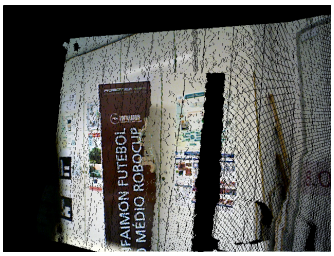
It was verified for some datasets that the graph built was with almost all images related directly to the reference image meaning the threshold used was not well calibrated, because for each image, a *considered* good rigid transformation was computed by the code immediately but was not a realistic good rigid transformation. The threshold was 50% of inliers and 0.01% of the points in the image were keypoints, this was too forgiving for some datasets so it would work for harder datasets, this parameters should have been more fine tuned.

VIII. Conclusion

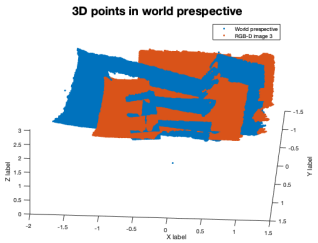
In this project it was intended to find the appropriate transformations between a set of images acquired by a depth and a RGB camera in order to superimpose them to reconstruct the photographic space and generate rigid transformations in order to move them to a reference camera perspective. In general, this was obtained with a positive level of quality, observed by the good overlap of the generated pointclouds for different images, although there was some

(a) Image 1 (reference)

(b) Image 3



(c) Transformation of image 3 to image 1



(d) Transformation of image 3 to image 1 (Pointcloud)

Fig. 11. Banner dataset examples

difficulty in some specific datasets, usually due to big depth ranges, big variations on the image perspective taken and few related points between compared images. Some improvements could be made, for example the possibility of implementing an Iterative Closest Point algorithm, which would bring more correct transformations, and, certainly, minor associated errors. The proposed solution was developed keeping in mind that the dataset is continuous, if this is not true the proposed algorithm is a slow one. The steps between consecutive images are usually small and, even when they are bigger, usually are close to some other close image so these pairs of related images will be quickly picked up. An interesting case is when the dataset is a sequence of photos in a straight line, never returning to the reference image and with some varying step sizes between images, in this particular case the dataset struggles.

We would evaluate this as a good solution for the proposed problem, it takes into account the properties of the given datasets to explore a solution with a good runtime and a good performance.

It's also important to mention that both RANSAC, Procrustes and our algorithm leveraging the tree structure were all implemented by us from scratch. We could have used functions from OpenCV or some other library and get better runtimes and, possibly, performance but the group thought it would be best to keep control of all the code we developed, this way there were no mysterious results.

List of Figures

References

[1] David G. Lowe *Distinctive Image Features from Scale-Invariant Keypoints*.