2 Good examples

1. Introduction and project definition

1.1. Kinect sensors

Kinect sensors were developed in order to enable users to control and interact with their computer or console through gestures without the need for a remote control. This is possible because they include two different types of cameras: a depth camera and a color camera, as it is represented in the following figure [1].

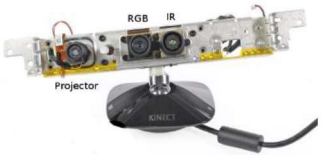


Figure 1. Kinect Sensor.

These cameras provide two types of images - depth and color images – that have the same resolution. When these images are properly combined, one is able to reconstruct the 3D scene of the world by assigning the RGB components of the pixels of the RGB image to the 3D pixels obtained by the depth image [1].

Figure 2. (at left) RGB image and (at right) corresponding depth image.

1.2. Problem definition

The main goal of this project is to build a 3D model of a scene and to track the Kinect sensor's trajectory in that scene, a problem that is known as the Self-Localization and Mapping (SLAM) problem.

Figure 3. 3D model of a scene (adapted from [2]).

Although the Kinect sensor provides a local representation of the environment, through the cloud of 3D points, this is not enough to reconstruct the scene from a single view since only local information is

available. Therefore, it is important to gather information from multiple views and align and fuse it into a single 3D model of the scene, assuming that the Kinect moves along a trajectory and acquires data at different positions and orientations.

In each position, one needs to compute the Kinect pose (position and orientation) with respect to the scene coordinates. Afterwards, one needs to fuse the 3D cloud of points obtained at the current position with the information obtained at previous acquisitions.

Thus, given a sequence of depth and RGB images, the tasks that have to be accomplished are the following:

1. Find a correspondence between point clouds, either by registering 3D point clouds or by using the RGB images to find point correspondences (feature matching);
2. Given pairs of (3D) corresponding points in two images, compute the transformation between them, i.e., the rotation and the translation;
3. Given transformations between pairs of images, propagate these transformations until all the point clouds are positioned in one single reference frame;
4. Filter the final point cloud.

1.3. Camera model

Each of the cameras referred above projects 3D points (given by $\boldsymbol{X} = [X\ Y\ Z]^T$) into an image plane (when the points are given by $\boldsymbol{x} = [x\ y]^T$), as it is represented below.

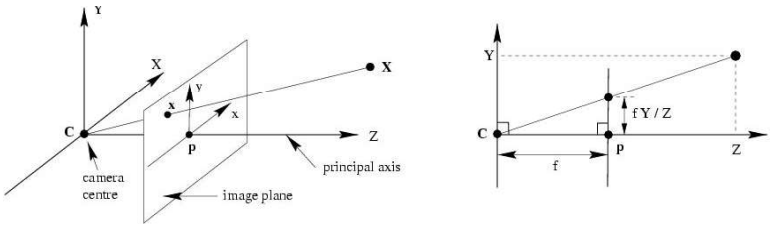


Figure 4. Camera model (adapted from [3]).

Considering a normalized camera (distance between the projection center and the projection point $f = 1$), the perspective projection is given by

$$x = \frac{X}{Z} \qquad y = \frac{Y}{Z}$$

or, in homogeneous coordinates, by

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \qquad \lambda \tilde{\boldsymbol{x}} = [I\ 0]\tilde{\boldsymbol{X}} = \boldsymbol{X}$$

The full camera model is derived having into account some:

- extrinsic parameters: the world frame is located at an arbitrary position.
- intrinsic parameters: focal length, scale factors, principal point.

Regarding the internal model, a new coordinate system is defined by a 2D coordinate transformation, where there is a conversion from metric coordinates to pixels

$$\begin{cases} x' = fs_x x + c_x \\ y' = fs_y y + c_y \end{cases}$$

where:

- $\boldsymbol{x} = [x\ y]^T$ is in metric units;
- $\boldsymbol{x}' = [x'\ y']^T$ is in pixels;
- f is the focal length;
- c_x and c_y are the coordinates of the principal point in pixels;
- s_x and s_y are scale factors in the x and y directions, respectively, in pixel/m.

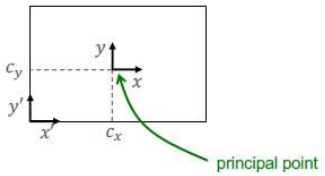


Figure 5. Internal model.

In homogeneous coordinates, one has:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} fs_x & 0 & c_x \\ 0 & fs_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \tilde{\boldsymbol{x}}' = K\tilde{\boldsymbol{x}}$$

where K is a upper triangular matrix that contains the intrinsic parameters.

Regarding the external model, a new coordinate system (world coordinate system) is defined by a 3D coordinate transformation

$$\boldsymbol{X} = R\boldsymbol{X}' + \boldsymbol{T}$$

where:

- $\boldsymbol{X}$ are the camera coordinates;
- $\boldsymbol{X}'$ are the world frame coordinates;
- $R \in \mathbb{R}^{3\times3}$ is a 3D-rotation matrix that expresses the rotation between the world and camera frames;
- $\boldsymbol{T} \in \mathbb{R}^3$ is a 3D-translation vector that represents the origin of the world coordinate frame expressed in camera coordinates.

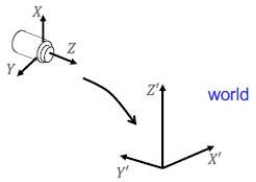


Figure 6. External model.

In homogeneous coordinates:

$$\tilde{\boldsymbol{X}} = \begin{bmatrix} R & \boldsymbol{T} \\ \boldsymbol{0}^T & 1 \end{bmatrix} \tilde{\boldsymbol{X}}'$$

Combining this expression with the perspective projection, one obtains:

$$\lambda \tilde{\boldsymbol{x}} = [R\ \boldsymbol{T}] \tilde{\boldsymbol{X}}'$$

Therefore, the final camera model is given by

$$\lambda \tilde{\boldsymbol{x}}' = K[R\ \boldsymbol{T}] \tilde{\boldsymbol{X}}' = P \tilde{\boldsymbol{X}}'$$

where P is a 3x4 matrix, denoted as camera matrix.

1.4. Keypoints

When one analyses a RGB image, only some pixels contain useful and relevant information – the keypoints. They are spatial locations or points in the image that define what is interesting or what stands out in the image, whose neighborhood should contain intensity variations in more than one direction. They are usually associated to fast changes of the gradient direction [4].

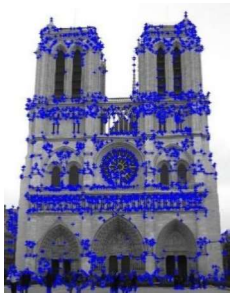


Figure 7. Keypoints of an image (adapted from [5]).

The reason why these points are special is because no matter how the image changes (rotates, shrinks or is subject to distortion), one should always be able to find the same keypoints in the image. That is why they are so important when doing a 3D reconstruction. When the Kinect moves from one position to another, the keypoints that are found in an image captured at the first position can be also found in an image captured at the second position. Therefore, if these keypoints are correctly matched, one is able to compute the Kinect's position and orientation at each instant and therefore join the information from both images.

1.5. Scale-Invariant Feature Transform

The Scale-Invariant Feature Transform (SIFT) is an algorithm that extracts and describes keypoints from an image. After the detection of the local features in the image, a 16x16 neighborhood around each keypoint is taken. It is divided into 16 sub-blocks of 4x4 size. For each sub-block, 8 bin orientation histograms are created. Therefore, a total of 128 bin values are available and are represented as a vector [4].

The SIFT algorithm was patented by the University of British Columbia and published by David Lowe in 1999 [6]. It can robustly identify objects even among clutter and under partial occlusion, because the SIFT feature descriptor is invariant to uniform scaling, orientation, and partially invariant to affine distortion and illumination changes.

1.6. Hungarian Method

The Hungarian method is an algorithm that solves the problem of keypoint matching in polynomial time. It was published in 1995 by Harold Kuhn [7] and reviewed later in 1957 by James Munkres. Therefore, the algorithm has been known since then as the Kuhn-Munkres algorithm.

The algorithm begins with the creation of a matrix whose entries represent the Euclidean distances between the descriptors of keypoints of different images. Afterwards, the algorithm performs the following steps [8]:

1. Subtract the smallest entry in each row from all the entries of its row.
2. Subtract the smallest entry in each column from all the entries of its column.
3. Draw lines through appropriate rows and columns so that all the zero entries of the cost matrix are covered and the minimum number of such lines is used.
4. Test for Optimality:
   (i) If the minimum number of covering lines is n, an optimal assignment of zeros is possible and the algorithm finishes.
   (ii) If the minimum number of covering lines is less than n, an optimal assignment of zeros is not yet possible. In that case, one proceeds to Step 5.
5. Determine the smallest entry not covered by any line. Subtract this entry from each uncovered row, and then add it to each covered column. Then, return to Step 3.

Figure 8. Keypoint matching (adapted from [9]).

1.7. Random Sample Consensus

When the keypoints have already been matched, one can estimate a geometric transformation between both sets of keypoints, i.e. estimate the rotation matrix R and the translation vector T. The Random Sample consensus (RANSAC) is an iterative method to estimate the parameters of a mathematical model from a set of observations that contains outliers (observations that should not influence the values of the estimates). The RANSAC algorithm works by identifying the outliers in a data set and estimating the desired model using data that does not contain outliers [10].

The RANSAC algorithm is composed of the following steps [10]:

1. Choose the model to fit to the data.

   For the case under analysis, one uses the model $p' = Rp + \boldsymbol{T}$.

2. Randomly select a sample subset containing the minimum number of observations to determine the model parameters from the input dataset.

   For the case under analysis, one needs at least 4 pairs of keypoints.

3. Fit the model to the selected subset (computation of model parameters using only the elements of the sample subset)

   For the case under analysis, one computes R_i *and* T_i *through the Procrustes problem, where i represents the iteration number.*

4. Check which elements of the entire dataset are consistent with the calculated parameters. An element will be considered as an outlier if it does not fit the model within some error threshold that defines the maximum deviation attributable to the effect of noise. Determine the number of inliers.

   For the case under analysis, one computes the error as $e = ||p'_k - R_i P_k - T_i||$, *where k represents the k-th pair of points. If* $e > Threshold$, *the pair of points will be considered an outlier. Otherwise, if* $e < Threshold$, *the pair of points will be considered an inlier.*

5. Repeat the Steps 2, 3 and 4 for a prescribed number of iterations.

6. The estimated model is reasonably good if sufficiently many points have been classified as inliers. Therefore, find the model that generated the maximum number of inliers, and use those inliers to re-estimate the model.

1.8. Procrustes problem

A Procrustes problem is a method that can be used to determine the optimal rotation for the Procrustes superimposition of an object with respect to another [11]. This algorithm assumes that there exist two sets of corresponding points $\{p_i\}$ and $\{p_i'\}$, with $i = 1, \dots, N$, where N is the number of pairs of points, such that they are related by

$$p'_i = Rp_i + \boldsymbol{T} + \boldsymbol{V}_i$$

where R is a 3x3 rotation matrix, $\boldsymbol{T}$ is a 3D translation vector and $\boldsymbol{V}_i$ a noise vector.

In order to solve for the optimal transformation R, T that maps the set $\{p_i\}$ onto $\{p_i'\}$, one needs to minimize a least squares error criterion given by:

$$\sum_{i=1}^{N} \|p'_i - Rp_i - \boldsymbol{T}\|^2$$

As a consequence of the least-squares solution to the previous equation, the point sets $\{p_i\}$ and $\{p_i'\}$ should have the same centroid. Therefore

$$\bar{p'} = \frac{1}{N}\sum_{i=1}^{N} p_i' \qquad p_c' = p_i' - \bar{p'}$$

$$\bar{p} = \frac{1}{N}\sum_{i=1}^{N} p_i \qquad p_c = p_i - \bar{p}$$

and the previous equation can be rewritten and reduced to

$$\sum_{i=1}^{N} \|p_c' - R p_c\|^2$$

Minimizing this equation is also known as the orthogonal Procrustes problem, which is a SVD based solution [11]. After the determination of R through SVD, the optimal translation vector can be determined as

$$T = \bar{p'} - R\bar{p}$$

2. Implemented algorithm

The algorithm to build the 3D model of a scene and to track the Kinect sensor's trajectory in that scene, given a set of depth and RGB images, was implemented in MATLAB.

2.1. Main script

At first, there is a script (**testproj.m**) that initializes all input variables, namely:

- the 3x3 matrixes with the cameras' intrinsic parameters (**K_depth** and **K_rgb**);
- the 3x3 rotation matrix and the 3x1 translation vector that allow the transformation of 3D coordinates represented in the depth camera reference frame to the RGB camera frame (**Rdtrgb** and **Tdtrgb**);
- the strings with the path of the files with depth and RGB data for each image (included in the array of structures **image_names**).

and calls the ***reconstruction*** function in order to perform the reconstruction with those variables, through the command:

```
[pcloud,transforms] = reconstruction(image_names,K_depth,K_rgb,Rdtrgb,Tdtrgb);
```

In the end, this function returns:

- one Nx6 matrix with the 3D points and RGB data of each point, represented in the world reference frame (depth camera coordinate system of the last image of the set) (**pcloud**);
- an array of structures where each element contains the transformation (rotation matrix and translation vector) between the depth camera reference frame and the world reference frame for each image (**transforms**).

Afterwards, the point cloud is:

- created through the MATLAB function ***pointCloud***;

- filtered through the MATLAB function ***pcdenoise*** in order to remove some outliers;
- downsampled using a box grid filter of size 0.005, through the MATLAB function ***pcdownsample***;
- displayed through the MATLAB function ***showPointCloud***.

2.2. Reconstruction function

The ***reconstruction*** function performs the following tasks:

- Load depth and RGB images;
- Compute the 3D coordinates of each point from each depth image, using the depth camera intrinsic parameters, through the ***get_xyz*** function;
- Align each pixel of each RGB image with the correspondent depth image, through the **get_rgbd** function. This function returns:
  - a virtual image whose pixels are aligned with the depth image pixels;
  - the indexes of the pixels of the original RGB image where the RGB components of each pixel of the depth image can be retrieved.
- For each pair of sequential images:
  - Extract keypoints and descriptors from the RGB images using SIFT, with a peak threshold equal to 10;
  - Compute Euclidean distance for each pair of descriptors;
  - Apply the Hungarian method to match the SIFT descriptors;

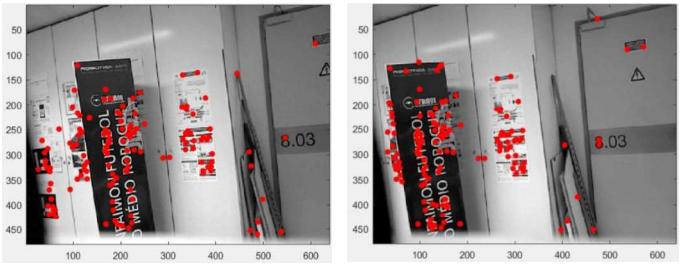


Figure 9. Example of matched keypoints from two images of the dataset *newpiv2*.

  - Apply RANSAC for 100-1000 iterations considering an error threshold of $\sqrt{0.025}$ in order to eliminate the bad matches and compute the best rotation matrix and translation vector between the depth camera reference frame of the first image and the depth camera reference frame of the second image.
- Propagate the transformations between pairs of images until all the point clouds are positioned in one single reference frame (depth camera coordinate frame of the last image).

2.3. RANSAC function

The ***RANSAC*** function receives as inputs:

- the minimum number of iterations of the method (**Iterations**);

- the threshold of the error in order to classify each pair of matched points as inliers or outliers (**Threshold**);
- the 3D coordinates of the points of each image, obtained from the depth images (**xyz1** and **xyz2**);
- the keypoints of each RGB image (**KeyPoint1** and **KeyPoint2**);
- the matches obtained by the Hungarian algorithm (**Matching Points**);
- the size of each image (**sizemat**);
- the indexes of the pixels of the original RGB images with the RGB components of each pixel of the depth images (**rgb_inds1** and **rgb_inds2**).

Afterwards, the function proceeds with the following steps:

- Get the 3D coordinates of the keypoints that were matched;
- While the number of iterations performed is lower or equal to the minimum one (**Iterations**) or the number of inliers in lower than 1/3 of the number of matching pairs and the number of iterations performed is lower or equal to 1000:
    - Randomly select 4 pairs of points from the set of matched points;
    - Computation of model parameters (rotation matrix and translation vector) using only the elements of the sample subset, through orthogonal Procrustes problem based on SVD;
    - Calculate error for each pair of points from the input dataset (matched keypoints);
    - Find inliers (pairs of points whose error is lower than the defined threshold);
    - Save the number of inliers and corresponding transformation.
- Find the transformation that generated the maximum number of inliers, find those inliers and use them to calculate the final rotation matrix and translation vector for that pair of images, which are delivered as outputs by the function.

3. Experimental results

After applying the previous algorithm to different datasets, the main results can be presented.

3.1. Dataset sofa

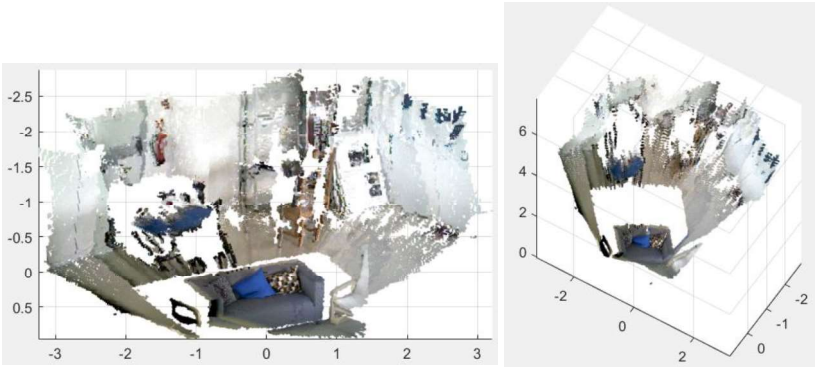


Figure 10. 3D reconstruction of a scene from images of the dataset *sofa*.

3.2. Dataset newpiv1

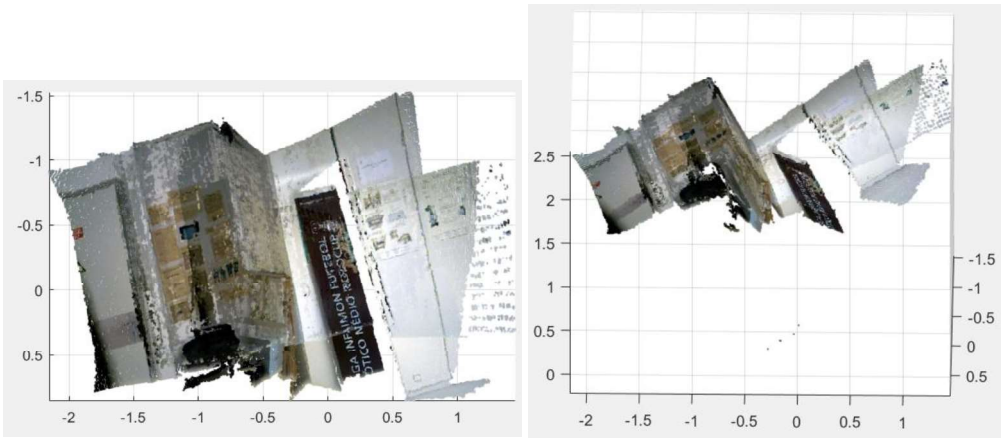


Figure 11. 3D reconstruction of a scene from images of the dataset *newpiv1*.

3.3. Dataset newpiv2

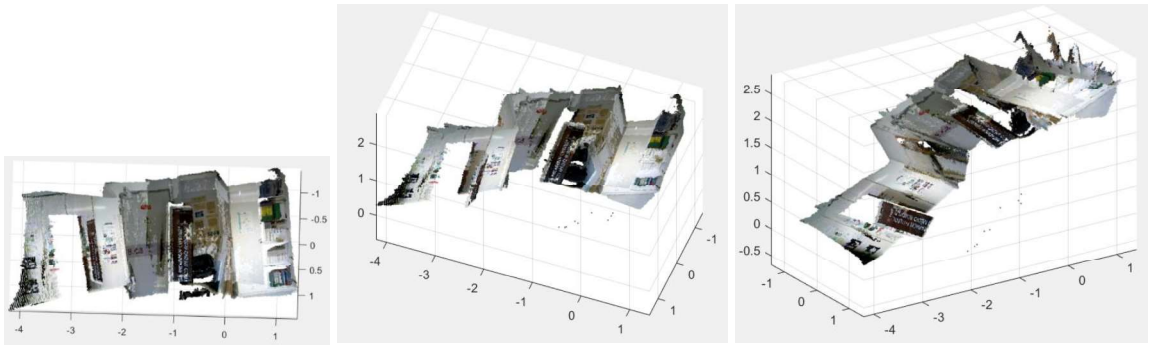


Figure 12. 3D reconstruction of a scene from images of the dataset *newpiv2*.

3.4. Dataset newpiv3

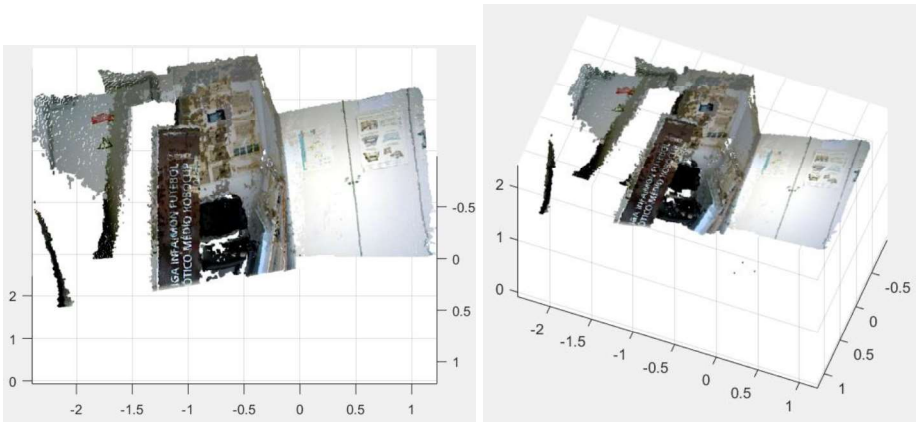


Figure 13. 3D reconstruction of a scene from images of the dataset *newpiv3*.

A lot of data, no comments! errors?, good/bad areas

3.5. Dataset newpiv4

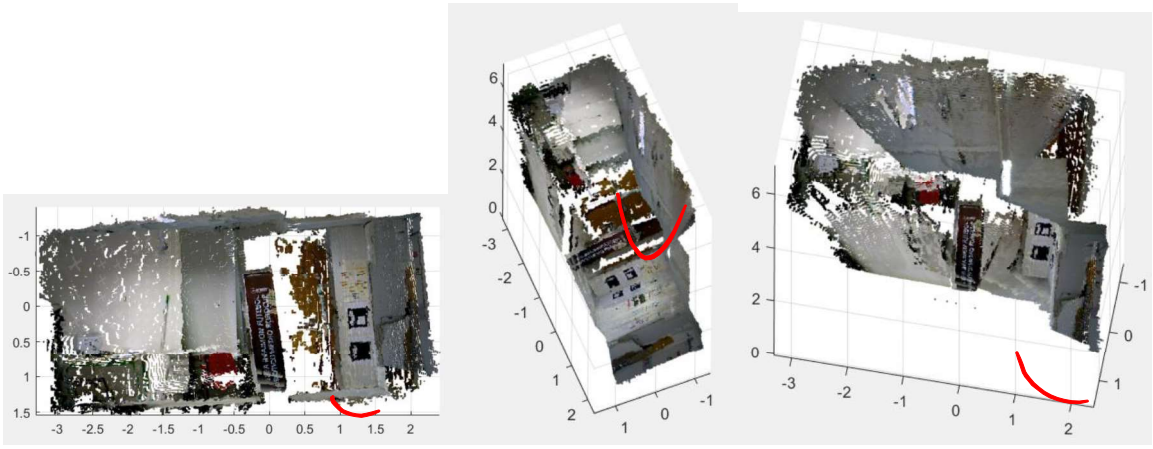


Figure 14. 3D reconstruction of a scene from images of the dataset *newpiv4*.

3.6. Dataset labpiv

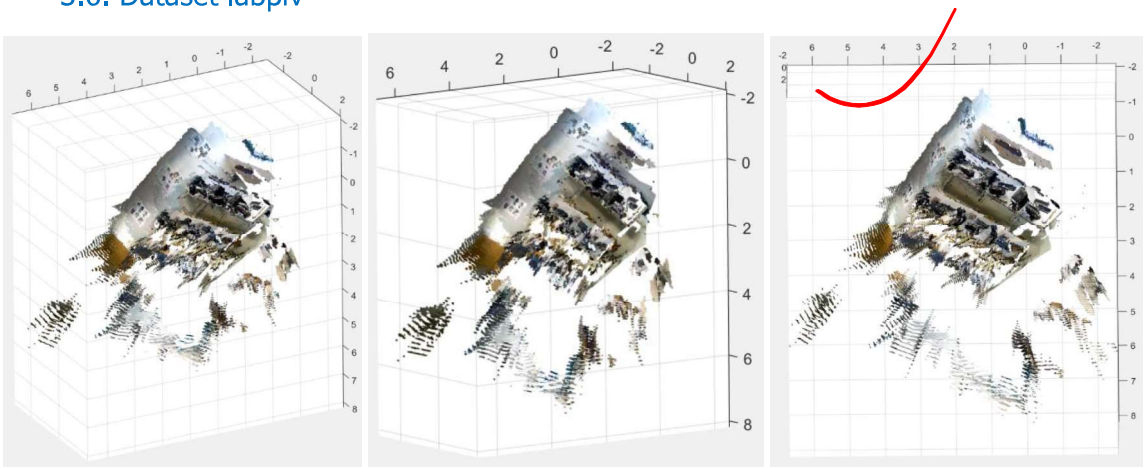


Figure 15. 3D reconstruction of a scene from images of the dataset *labpiv*.

3.7. Dataset labpiv_malandro

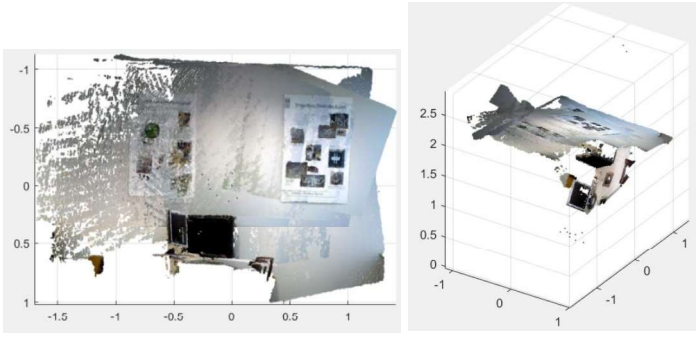


Figure 16. 3D reconstruction of a scene from images of the dataset *labpiv_malandro*.

Good ... some analysis? Did it fail? Tunning!

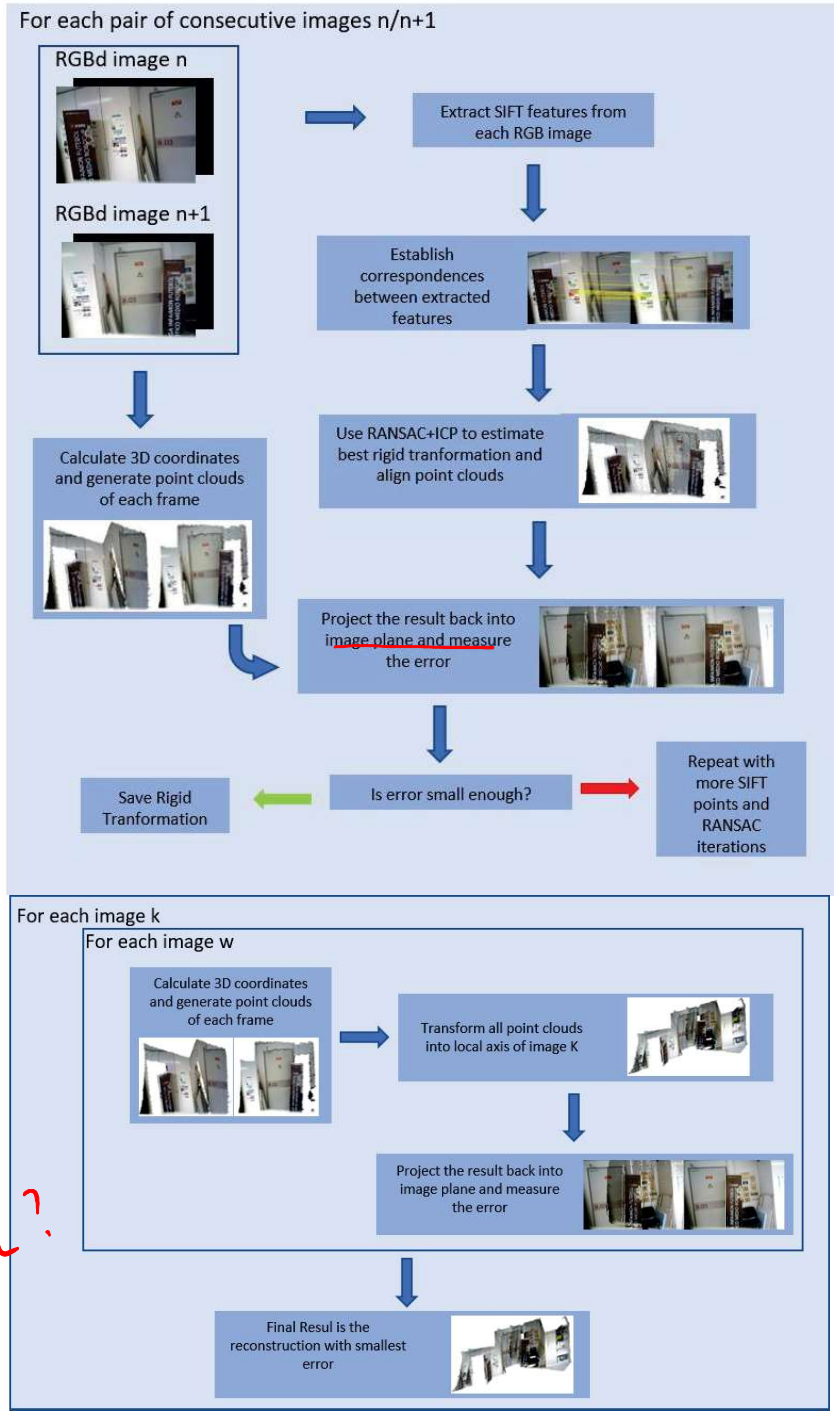


Figura 1 – Algorithm Overview

2.1. RBG and Depth Images Correspondence

Cameras project 3D points into the image plane. This transformation is not invertible unless we have additional information about the point location in space (e.g., depth, which is the case). Let (X, Y, Z) be the coordinates of the point P in the camera frame and let (u, v) be the coordinates of the projected point p on the image plane.

The transformation which maps 3D points into image points is called the perspective projection and is given by the relation (1).

$$u = -f * \frac{X}{Z}$$
$$v = -f * \frac{Y}{Z} \quad (1)$$

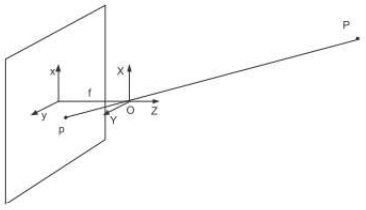


Figure 1 – Perspective projection.

The normalized camera model in homogeneous coordinates is then given by (2).

$$z\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \Pi_0 \vec{X} \quad (2)$$

To couple the RGB and depth information, which are captured by two different cameras, a rigid transformation between the frames is required, since the optical centre of both cameras is not coincident. The rigid body transformation between the frames can be written in homogeneous coordinates by (3),

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} R & T \\ \vec{0} & 1 \end{bmatrix} \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \\ 1 \end{bmatrix} = G\overrightarrow{X_0} \quad (3)$$

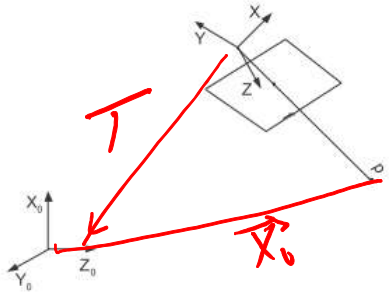


Figure 2 – Extrinsic Parameters

where $X_0 \in P^3$ is the coordinates in the reference frame, $R \in R^{3x3}$ is a rotation matrix and $T \in R^3$ is a translation. The parameters of the transformation (R, T) are called the extrinsic parameters of the camera, since they depend on the camera location and orientation with respect to the reference frame.

The rotation matrix is an orthogonal matrix ($R^T R = I$) and it preserves the orientation of the axis ($det(R) = 1$).

Since image points are measured in pixels and the image frame may not be centred a final transformation is required to make the conversion of metric to pixel coordinates. It is also indirectly responsible for the image discretization (4).

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} fs_x & fs_\theta & o_x \\ 0 & fs_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = K\vec{x} \qquad (4)$$

The transformation matrix becomes an upper triangular matrix where the parameters ($fs_x, fs_y, fs_\theta, o_x, o_y$) are denoted as intrinsic parameters of the camera since they only depend on the camera and are independent of the camera position and orientation. Matrix $K \in \mathbb{R}^{3\times3}$ is called the matrix of intrinsic parameters.

The camera model with intrinsic and extrinsic parameters is given by (5) where $\Pi \in \mathbb{R}^{3\times4}$ is a 3×4 matrix denoted as the camera matrix and performs the projective transformation.

$$\lambda\overrightarrow{x'} = K[R\ T]\overrightarrow{X_0} = \Pi\overrightarrow{X_0} \qquad (5)$$

Since depth information is provided, we can recover 3D coordinates for each pixel applying the inverse projective transformation (using the inverse transform of Π in 5). In fact this is done in the function *"get_xyzasus"*, using equations (1) solved in order to X and Y. To align color with depth images in the same camera plane, a rigid transformation is performed in the function *"get_rgbd"*.

2.2. Image Alignment

To estimate the rigid transformation that relates each pair of consecutive images, we used SIFT algorithm to identify keypoints in those images and then perform RANSAC to find the inliers of the most voted transformation. With those inliers, we find the best transformation in a least squares sense. To perform SIFT on the images we used *"vl_sift"* from VLFeat toolbox. The use of SIFT is motivated by the robustness of this descriptor with changes in lighting, cluttering, scale and noise. Using this algorithm, we obtained two structures F and D. Each column of F represents a keypoint and has the format $[X; Y; S; TH]$, where X, Y is the center of the keypoint (which can have subpixel precision), S is the scale and TH is the orientation (in radians). Each column of D is the descriptor of the corresponding keypoint in F. A descriptor is a 128-dimensional vector with information about image gradient magnitude and orientation around the keypoint. The tune parameters explored, by order of sensibility, are shown in Table 1, and are automatically tuned in order to find a reasonable number of keypoints (200).

Table 1 – SIFT parameters range interval tuning.

SIFT Parameter	Range	Value used
peak threshold	[0 – 0.05]	Dynamic
edge threshold	[5 - 500]	50
number of octaves	[20 -128]	20

To ensure there are enough points to estimate a good transformation but not too many so that the calculation becomes slow, these values were tuned run-time to keep the number of points between 100 and 300.

After this, we needed to establish correspondences between keypoints in different images. For this we've used a more efficient implementation of the Hungarian algorithm designated Auction algorithm. This algorithm finds the assignment of pairs of points that minimizes some cost function. The cost function we used is calculated using 5 different parameters: (1) Manhattan distance between SIFT descriptors; (2) Manhattan distance of the points coordinates in the image; (3) absolute difference of color measured in channel H of HSV image encoding; (4) absolute difference in the local scale; (5) absolute difference of histogram orientation.

The values of each criteria were normalized to a zero mean and unitary standard deviation and used to compute the benefit value of the match matrix considering inverse of the sum of these parameters.

The algorithm used to find the matching keypoints was the Auction Algorithm which maximizes the goal function by applying several variations of a combinatorial optimization algorithm which solves assignment problems, and network optimization problems with linear and convex/nonlinear cost.

2.3. Rigid Transformation Computation

The determination of the rigid transformation between two point clouds is made knowing the 3D position of the matching keypoints. The formulation computes the translational and rotational components of the transform in closed form, as the solution to a least squares formulation of the problem using singular value decomposition (*SVD*) based on the standard [R, T] representation (orthogonal Procrustus problem).

Assuming two corresponded point sets $\{m_i\}$ and $\{d_i\}$, finding the rigid transformation (R, T) that best suits the sets can be achieved through the procedure presented in Table 2.

Table 2- Orthogonal Procrustus problem procedure.

Operation	Equation	*N°*

Rigid Transformation	$d_i = Rm_i + T$	*6*
Error Function to be Minimized in Least Squares Sense	$\Sigma^2 = \sum_{i=1}^{N} \left\| d_i - \hat{R}m_i - \hat{T} \right\|^2$	*7*
Mean Removal Translation Removal	$\bar{d} = \frac{1}{N}\sum_{i=1}^{N} d_i \qquad d_{ci} = d_i - \bar{d}$ $\bar{m} = \frac{1}{N}\sum_{i=1}^{N} m_i \qquad m_{ci} = m_i - \bar{m}$	*8*
Maximization of Correlation Matrix by SVD Decomposition	$H = m_{ci} d_{ci}^T = U \Lambda V^T$	*7*
Rotation Matrix	$\hat{R} = U \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & det(UV^T) \end{bmatrix} V^T$	*8*
Translation Vector	$\hat{T} = \bar{d} - \hat{R}\bar{m}$	*9*

Using a random subset of matching keypoints (4 points) this formulation is applied and R and T are determined. Then, one of the point clouds is transformed into the other's reference frame and keypoints are categorized into inliers and outliers based on the distance they land from their corresponding keypoint. If this distance is below 5 cm, they are considered inliers. After some iterations, the inliers of the most voted transformation are used to compute the final rigid transformation, considering the orthogonal Procrustus problem.

2.4. Error Evaluation

After estimating the transformation between both images of a pair, we perform an error checking stage, consisting in projecting the registered point cloud into the image plane and comparing with the original coordinates.

This method is based on the following intuitive reasoning: "If the world was like this, how would the pictures taken look like?". To do this, the whole point cloud is transformed into each camera local axis and then a projection is taken keeping only the closest point to the camera, just like in ray marching. The image generated can then be used to perform qualitative evaluation, or the same method can be used, without explicitly generating this image, to estimate the coordinates of the points that would be

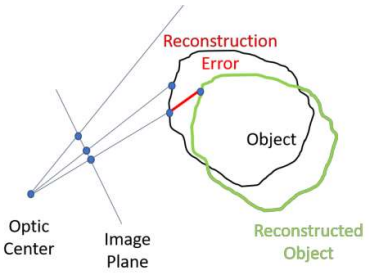


Figure 3 – Ray marching to generate camera projection and estimate average reconstruction error.

seen by the camera and then compare them to the real coordinates measured. The result is an error measure with a geometric interpretation: the average distance between the original and reconstructed points.

If this average error is above a threshold, the algorithm restarts for that image pair with more SIFT points and more RANSAC iterations, if it keeps larger than the limit after this, the algorithm terminates and returns the valid reconstructed point cloud.

2.5. Iterative Closest Point

Once the previous test is passed, ICP algorithm can be performed with some confidence that the estimated transformation is good enough so it won't be trapped by a local minimum or, even worse, tend to an inferior estimate. The error of the ICP is also considered and if it is above a certain threshold, the whole alignment is discarded and repeated with more SIFT points and RANSAC iterations.

2.6. Choice of Global Reference Frame

After determining the rigid transformations between pairs of images, the choice of the overall global reference frame is desirable in order to fuse all the local point clouds.

Choosing one of the local reference frame as the global reference frame was considered, since it would simplify the choice and resulting in just a cascade of the local rigid transformation matrices (in homogeneous coordinates) in the direct order (or its inverse matrix in the inverse order) presented in Table 3.

Table 3 - Computation of Global Rotational Matrix. Abuse of notation was considered in the intermediate step for pedagogic purposes.

Operation	Equation	*Legend*
Rigid Transformation (between local frames)	$P_1 = R_{21}P_2 + T_{21} = \widehat{K_{21}}P_2$ $P_2 = R_{32}P_3 + T_{32} = \widehat{K_{32}}P_3$	*10*
Global Reference Frame transformation matrices (1 or 3 respectively)	$P_1 = \widehat{K_{21}}\widehat{K_{32}}P_3$ $P_3 = \widehat{K_{32}}^{-1}\widehat{K_{21}}^{-1}P_1$	*11*

By measuring the reconstruction error when considering as global reference each one of the local frames, it is possible to obtain an estimate of the best global reference frame.

Intuitively, the best global reference frame should be one of the intermediate trajectory points since the transformations matrices will propagate the error and accumulate larger errors as the number of steps increases.

2.7. Downsample Point Cloud

When all transformations are estimated and the best global reference frame is determined, the local point clouds are transformed, but since point clouds overlap, the final result would have unnecessary information that can be discarded by applying a grid and keeping only a single data point for each voxel of that grid. That is achieved using the function *"pcdownsample"* with a grid of 0.5 cm to have a high quality visualization.

3. Experiments

To test the algorithm developed, some datasets were used and mean reconstruction error, ICP root mean squared error and resulting point cloud were used for evaluation.

Table 4 – Comparison of results obtained with different databases.

Final Point Cloud	Nº images	ICP mean error	Reconstruction mean error (cm)	Nº of reference image
	9	0.0029	2.1	7
	20	0.0037	6.1	14
	22	0.0039	5.4	2

	30	0.0037	19.3	9
	40	0.0081	40.5	9

Figure 4 – Example of a projection of an unsuccessful (left) and successful (right) reconstructions into a camera plane and the corresponding original image.

4. Conclusions

We can conclude that the algorithm is robust enough to estimate good transformations between each pair of images even when there are some changes in lighting. Despite this, when there are not a considerable similarity between consecutive frames, the algorithm struggles and it fails even using a very large amount of SIFT points and RANSAC iterations, which were the parameters with a larger impact on the quality.

Things You should not do!

- describe things you don't know
- formulas with errors
- Text that says nothing
- complicated words to explain precise "things" ... I interpret as "you don't know what you're talking about"

1. Introdução

Pretende-se neste trabalho desenvolver um algoritmo que permita reconstruir cenários em 3D a partir de dados obtidos por um Kinect. Este dispositivo dispõe de uma ~~*depth camera*~~ e de uma *color camera*, o que permite extrair dois tipos de imagens que têm a mesma resolução e estão alinhadas. Isto significa que o Kinect fornece informações sobre a profundidade e cor associados a cada pixel.

O conjunto de imagens fornecidas irá constituir um *dataset* que permitirá reconstruir, por exemplo, o modelo 3D de uma sala de aula. Estas imagens são retiradas de diferentes perspectivas de dentro da sala, sendo que em cada posição que o Kinect ocupa, este irá processar a sua posição e orientação.

Depois de obter estes dados, o algoritmo desenvolvido irá criar uma nuvem 3D de pontos (*point cloud*) do ambiente que o Kinect analisou.

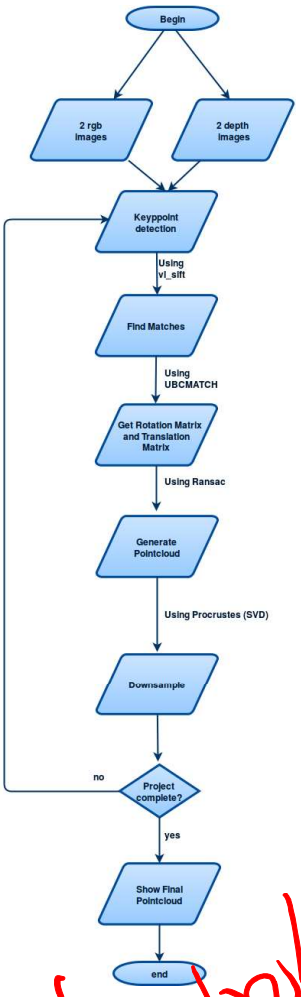
2. Especificação do Algoritmo Implementado

O algoritmo, cujo fluxograma pode ser observado à direita, tem como primeira etapa o processamento das imagens dos ambientes 3D que se pretendem reconstruir. Para tal, será necessário obter as coordenadas XYZ de cada pixel de cada imagem processada com base nas informações fornecidas pela *depth camera*. Estes dados serão posteriormente utilizados para obter as características RGB de cada um desses pixels.

De seguida, utilizam-se algumas funções da *toolbox* vlfeat v0.9.20 que servirão para fazer extracção de features das imagens do *dataset* em questão e consequente matching das mesmas. O objectivo é que estas funções detectem pontos chave comuns entre imagens obtidas a partir de ângulos diferentes. Esta extracção de *features* é ~~posteriormente~~ aprimorada através da função Ransac, que é responsável por seleccionar as features mais importantes de entre aquelas que já foram emparelhadas entre imagens diferentes.

A próxima etapa será o cálculo das matrizes de rotação que permitem juntar imagens sucessivas que não estejam alinhadas segundo o mesmo referencial. Isto é primeiramente feito para as duas primeiras imagens de cada dataset e seguidamente, é executado um ciclo *for* que repete as mesmas funcionalidades já descritas para as seguintes imagens.

No fim, é obtido um vector que é composto pelas componentes XYZ e RGB da *point cloud* que se pretende obter. Depois de processada a *point cloud*, é possível fazer um downsample que reduz o número total de pixels obtidos, de forma a que o processamento seja feito de forma mais rápida.



3. Implementação

3.1. Get_xyzasus

Esta função é de grande importância pois será responsável pela obtenção dos parâmetros XYZ de cada imagem processada, para que posteriormente com estas coordenadas XYZ seja obtida a *depth*.

As coordenadas X e Y podem ser facilmente encontradas a partir de uma transformação K_{IR} (à qual foi dado algum ênfase ao longo das aulas de laboratório) e que pode ou não ser uma transformação euclidiana, isto é, se os ângulos não variarem. Quanto à coordenada Z, esta pode ser determinada a partir da equação abaixo descrita:

$$Z = \frac{1}{(\alpha z_{IR} + \beta)}$$

Sendo que z_{IR} é o valor de IR nas coordenadas pretendidas, para obter os pixels basta resolver a transformação:

$$X_{RGB} = R * X_{IR} + T$$

Com o valor de X, Y conseguimos ir buscar o valor de Z através da posição do centróide.

3.2. Get_rgbd

Esta função será responsável pela obtenção dos parâmetros RGB de cada imagem processada. Para tal, serão necessários os parâmetros intrínsecos da rgb_camera, bem como o conjunto de pontos obtidos anteriormente através de get_xyz. A função também precisará de Rdtrgb and Tdrgb como inputs de forma a poder transformar as coordenadas XYZ representadas na *depth_camera* para o referencial RGB.

Sendo assim, serão obtidas as componentes RGB das coordenadas XYZ de cada imagem que for processada. Isto será particularmente útil para que seja feita a point cloud do ambiente 3D que se pretende reconstruir.

3.3. Toolbox Vlfeat v0.9.20

Para implementar o algoritmo descrito na secção anterior, procedeu-se à utilização da *toolbox* vlfeat v0.9.20, uma ferramenta que implementa algoritmos de processamento de imagem, especializando-se em análise de imagens, extracção local de *features* e *matching* dessas mesmas *features*. Mais especificamente, as funções desta *toolbox* que foram utilizadas são vl_sift e vl_ubcmatch, que passarão a ser explicadas a seguir.

A função vl_sift recebe como input principal uma imagem em escala de cinza com *single precision* e terá como output dois vectores - o vector F, em que cada coluna corresponde a um *feature frame* extraído da imagem de input e o vector D que consiste

num *descriptor* do *frame* correspondente em F. Para melhorar a precisão da extracção de *features*, decidiu-se utilizar dois parâmetros de inputs adicionais que actuam como filtros - PeakThresh e EdgeThresh. Peak Threshold corresponde à quantidade mínima de contraste para que seja aceite um *keypoint* (*feature*). Quanto maior o valor deste parâmetro, menor será o número de features. Edge Threshold corresponde a um factor de rejeição de *edges*. Quanto menor o valor deste parâmetro, menor será o número de features.

A função ubc_match será responsável pelo *matching* de features de duas imagens diferentes. Isto será particularmente útil para criar novas point clouds, uma vez a função detecta quais os conjuntos de pixels que são comuns entre as duas imagens. Esta função recebe como dois inputs os *descriptors* obtidos na função vl_sift e ainda dispõe de um parâmetro de *threshold*. O output serão as features que foram avaliadas como sendo comuns entre as duas imagens (vector Matches) e as distâncias euclidianas ao quadrado entre os matches (vector Scores).

3.4. Rotação e Translação

Após obtido o vector *Matches*, está-se perante dois conjuntos de pontos correspondentes: P_1 no referencial Ref_1 (conjunto de pontos da imagem 1) e P_2 no referencial Ref_2 (conjunto de pontos correspondentes da imagem 2). P_1 e P_2 têm naturalmente a mesma dimensão $N*3$ (ou $3*N$). ***Nota:*** Estes pontos P_1 e P_2 são apenas os pontos correspondentes, é portanto necessário utilizar a expressão que se segue para calcular todos os outros pontos.

Pretende-se encontrar uma matriz de rotação e um vector de translação que permita transformar P1 para o referencial Ref_2.

$$P_1 = R_{12} * P_2 + T_{12}$$

$$T_{12} = P_1 - P_2 * R_{12}$$

Dada esta expressão ser iterativa (recursiva), uma fórmula geral seria:

$$P_N = R_{N\,N+1} * P_{N+1} + T_{N\,N+1}$$, **Com N > 0**

Primeiro obtém-se R_{12} com o auxílio da função de matlab SVD que aplica o algoritmo Procrustes. O funcionamento de SVD encontra-se demonstrado abaixo:

$$A = U\Sigma V^T$$

Em que U é uma matriz ortogonal $(M * M)$, Σ é uma matriz diagonal $(M * N)$, V^T e é uma matriz ortogonal $(N * N)$. Esta decomposição, consiste em transformar a matriz A num produto de matrizes, o que equivale a dizer, realizar uma rotação, translação e novamente uma rotação.

Fazendo:

$$A\,A^T = V(\Sigma\,\Sigma^T)V^T$$

$(\Sigma\,\Sigma^T)$ λ *para* $A\,A^T$ *e* σ^2 *para* A *(contém os valores próprios para* $A\,A^T$*)*

(V) *Contém os vectores próprios para* $A\,A^T$

O que se pretende é fazer $|min| = \|A - BC\|_k^2$, em que a característica da matriz B é simultaneamente igual à característica da matriz C e a k. Ou seja, isto tudo, para provar que o SVD realiza a aproximação de uma matriz através de outra matriz de característica inferior.

A um nível mais experimental, após terem sido encontrados os pontos dos matches, centra-se esses pontos e calcula-se a matriz de rotação e a matriz de translação (esta última, implicitamente) com o algoritmo SVD. Estes pontos são apenas os correspondentes. De seguida é só aplicar a matriz de rotação à totalidade dos pontos. ***Exemplo:*** Os pontos da pointcloud 1 terão que ser transformados no referencial 2, utilizado a matriz de rotação R_{12}.

3.5. Ransac (Random Sample Consensus)

Após a utilização da função vl_ubcmatch, são extraídos apenas os pontos correspondentes entre as duas imagens, denominados pelos vetores u e v. Como explicado anteriormente, estes pontos são obtidos com base numa análise localizada em torno de cada pixel proveniente da função vl_feat (que retorna apenas *key points* ou *features* e o correspondente descritor).

Nesta altura é provável que surjam dois problemas:

- Primeiro, pode acontecer que um destes pontos que supostamente é correspondente, na realidade não seja (por exemplo, por existirem vários pontos com descriptores semelhantes). Basta uma correspondência errada para que toda a transformação posteriormente calculada esteja totalmente errada.
- Segundo, podem existir zonas com muitos detalhes que mudam de local ou existir zonas que sejam efectivamente iguais em lugares diferentes. Estes fenómenos provavelmente irão "baralhar" o programa.

O algoritmo iterativo Random Sample Consensus aparece no sentido de resolver estes dois problemas com a seguinte ideia base: testar uma quantidade de modelos construídos a

partir do mínimo de pontos necessários (4) escolhidos aleatoriamente, e escolher o candidato que melhor se adequa ao maior número de pontos.
O algoritmo Ransac implementado no projecto em causa pode ser esquematizado nos seguintes pontos:

1. Escolher 4 pontos correspondentes das duas imagens rgb;
2. Com base nestes pontos calcular a matriz de rotação e translação 3D;
3. Contar quantos pontos "encaixam" no modelo calculado (inliers). Assume-se que um ponto é inlier se a distância entre o ponto calculado e o original for inferior a 2cm (que corresponde ao parâmetro Dif, escrito dentro da função);
4. Repetir o processo 3000 vezes guardando sempre os pontos correspondentes ao modelo que teve mais inliers;
5. Retornar apenas os pontos que validaram o melhor modelo para que posteriormente possa ser calculada a matriz de rotação e translação.

Nota: no ponto 3 do ransac utilizou-se como critério a distância de 2 cm para os dois primeiros sets de imagens, porém para os *dataset3* e *dataset4* foi necessário alargar este distância para 7cm.

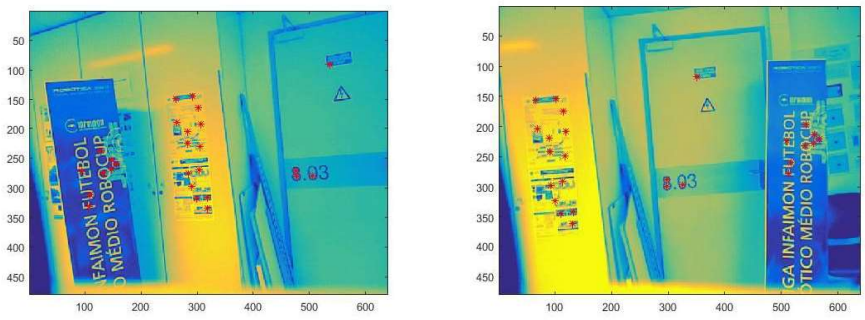


Figura 1 - Imagens e respectivas *features* antes de se utilizar a função Ransac

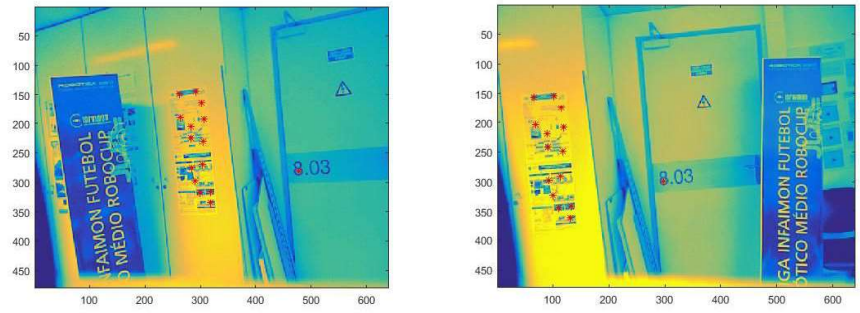


Figura 2 - Imagens e respectivas *features* depois de se utilizar a função Ransac (com parâmetro dif = 0.007)

$$\text{Cost}_f = e^{\frac{\left(\cos^{-1}(\text{Cost})\right)^2}{\sigma}} \quad (5)$$

After preparing filtering the cost function the Hungarian Algorithm is applied and the matches given.

2.4. RANSAC

After performing feature matching, the RANSAC algorithm is executed. It's an iterative method for estimating parameters of a mathematical model. For that uses random chosen points to be used in several tests and choose the ones with less error.

First of all, a model for the algorithm should be chosen. For this case, the rigid body transformation (6) was selected, since we have 2 sets of image points and want to discover a rotation (R) and translation (T) matrix. R and T are unknowns and Xi, Yi and Zi are N world features for set 1 and 2, previously detected.

$$\begin{bmatrix} X_i^{(2)} \\ Y_i^{(2)} \\ Z_i^{(2)} \end{bmatrix} = R \begin{bmatrix} X_i^{(1)} \\ Y_i^{(1)} \\ Z_i^{(1)} \end{bmatrix} + T \quad , i = 1, \ldots, N \quad (6)$$

Since we have 3 equations, we can select 4 points to estimate R and T, 3 points for translation and 1 for rotation.

RANSAC choose 4 random points in each iteration, for which a matrix R and T are going to be estimated. This is called the *Procrustes* Problem, the problem of finding a transformation that more closely describes the relationship between both world sets.

The problem formulation is represented in (7), which is basically a minimization of the least square error of the rigid body transformation.

$$R^*, T^* \arg\min \left\| \begin{bmatrix} X_i^{(2)} \\ Y_i^{(2)} \\ Z_i^{(2)} \end{bmatrix} - R \begin{bmatrix} X_i^{(1)} \\ Y_i^{(1)} \\ Z_i^{(1)} \end{bmatrix} + T \right\|^2 = R^*, T^* \arg\min \sum \left\| P_i^{(2)} - R P_i^{(1)} + T \right\|^2 \quad (7)$$

An alternative way of representing (6) is (8).

$$\sum_{i=1}^{N} p_i^{(2)} = \sum_{i=1}^{N} (R p_i^{(1)} + T) = \sum_{i=1}^{N} R p_i^{(1)} + T \quad (8)$$

Dividing both sums by N, results the centroid equation (9).

$$\overline{P^{(2)}} = R \overline{P^{(1)}} + T \quad (9)$$

If we subtract equation (6) for equation (9), i.e. removing the average of point clouds, will result the equation (10) which relation is only a rotation matrix.

it applies this transformation to X1, in order to get X2_estimate, and measures the error between the estimated points and the real ones, check equation 1 .

$$error = X2\_estimate - X2 \tag{1}$$

In the following iterations, a new subset is considered and the current error is compared with the previous one. When the loop reaches the end, the points that will be taken into account will be the ones that correspond to the least error, which will correspond to the inliers.

2.4 Transformation Matrix

With the set of inliers (matches) defined, it is possible to calculate the transformation matrix between two sets of points.
The goal is two calculate each transformation matrix that will transform the points' coordinates in a current frame into points' coordinates in the base frame. Equation 2 represents the transformation matrix between the current frame n and the base frame, where p represents the number of intermediate frames between the base and current frame.

$$T_{(1)(n)} = T_{(1)(n-p)} ... T_{(n-2)(n-1)} T_{(n-1)(n)} \tag{2}$$

$$T = \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \tag{3}$$

Equation 3 represents a general transformation matrix, that in this specific case will have to be an affine transformation. An affine transformation preserves points, straight lines and planes. In this case, we are dealing with rigid transformations and consequently it is necessary to satisfy the following condition: the determinant of the transformation matrix has to be equal to zero.

2.5 Fuse point clouds

To reconstruct a 3D scene it is necessary to fuse all point clouds. Therefore, the following steps are required:

1. Determine the point clouds for each set of images;
2. Define a base frame: the frame of the first image;
3. Determine the transformation matrix between pairs of sequential images, which requires the correspondent points between a set of two images;
4. Transform the coordinates of the points of each image into the coordinates of the base frame;

Our solution

As described in the introduction there are several sub-problems that needs to be solved before we are finally able to construct a 3D model from a set of images. Each sub-problem will be described here along with explanations of the theoretical concepts that have been used in the implementation.

Fusing depth and RGB images

By now we are now aware that the Kinect is equipped with two cameras; an RGB camera and a depth camera. Combining two images, one taken by the RGB camera and one by the depth camera (at the same time), one can create a 3D image. The 3D-image will be in the form of a point cloud; a set of points with three dimensional coordinates and an assigned color value. Creating a point cloud from a depth and an RGB image is an important sub-problem of the project. The two images contain all the information needed to create a point cloud. The image taken by depth camera contains information about the distance from each point shown in the image to the camera; the depth. The RGB image contains information about the color of each point. To compare and combine points from images taken by the two cameras the intrinsic and extrinsic camera parameters are required. These concepts will be explained in the next subsections.

Extrinsic camera parameters

Each camera uses its own coordinate system, in which the position of each pixel of an image is defined. To be able to compare points in images taken by two different cameras, it is important align the coordinate systems of the two cameras. For the Kinect the coordinate system of the depth camera will be used as the reference frame. Therefore, the points given in the RGB coordinate frame have to be transformed into the reference frame. This transformation is given by a rotation; giving the orientation between the cameras, and a translation; giving the location of the cameras relative to each other.

$$\begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ 1 \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \quad (1)$$

Equation 1 describes how the coordinates of a point can be transformed from one coordinate frame to another by using the rotation and translation between them. The matrix of r's is called the rotation matrix, $\mathbf{R}$, and describes the rotation between the frames. The vector consisting of $t_{1,2,3}$ represents the translation, $\mathbf{T}$, from one frame to the other. The rotation and translation between the depth camera and the RGB camera of the Kinect is given. Calculating the coordinates of the pixels from the RGB image in the reference frame is simply done by inserting each x,y-coordinate pair into the equation.

Intrinsic camera parameters

The next issue related to matching pixels in images taken by different cameras is the difference of scale. The pixels in one camera might have a different size and center than those in another camera. The intrinsic parameters are needed to correct for these differences.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} fs_x & 0 & c_x \\ 0 & fs_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \tag{2}$$

Equation 2 describes how to calculate the x- and y-coordinates in metric units from the pixels in an image. $p_{x,y}$ are the x- and y-coordinates given in pixels. f is the focal length of the camera, and is dependent of properties of the camera lens. $s_{x,y}$ is the pixel sizing, describing the resolution of the image. $c_{x,y}$ are the coordinates of the principal point, usually in the center of the image. These parameters are unique for each camera. From now on, the matrix containing the intrinsic parameters will be referred to as **K**. We are given **K** for each of the two cameras. This simply leaves us with using eq. 2 to calculate the metric x- and y-coordinates of the points in each image.

Producing a point cloud from a pair of images

The techniques explained above have given us the tools we need to successfully match points in two images to produce a 3D-image; more precisely a point cloud. This section will explain how to go from a depth image with a corresponding RGB image, to a full point cloud representation of the image. First, the depth value of each point in the depth image, is used to assign the z-coordinate of each x-y pair. Once the z-values are assigned, the metric coordinates of x and y are found, with help from the intrinsic of the depth camera.

Now, the RGB image is transformed into the frame of the depth image by translating and rotating it. Using the intrinsic parameters of the RGB-camera, one can finally find which points in the RGB image corresponds to those given in the depth image, by comparing the metric x- and y- coordinates. At this point, color values are assigned to the points in the depth image, based on the value of the RGB image in each point. This set of three dimensional points form a point cloud, representing the 3D-model of the image.

Creating a point cloud from a set of image pairs

SLAM

The purpose of this project is not only to create a single point cloud from one pair of a depth and RGB image, but to simulate self-localization and mapping for e.g. a robot. With a sensor as advanced as the Kinect we are able create a three dimensional map, as opposed to a laser, which can only create a 2D map. By using a series of images taken from different views in a given scene, it can simulate a robot moving around in the scene, taking images with the Kinect as it moves. Our MATLAB function will process

Introdução

O presente trabalho é sobre a reconstrução de uma figura 3D a partir de um ”dataset”, mais concretamente, de um conjunto de imagens (cada uma decomposta em duas outras imagens, uma RGB e outra Depth).

São objetivos deste trabalho conseguir desenvolver e entender um método que nos permita executar tal funcionalidade para qualquer ”dataset” de imagens.

Está organizado em seis partes. No capítulo um, será abordada a maneira como decidimos estruturar o problema no matlab. No capítulo dois, optámos por abordar a maneira de reconhecer pontos que possam ser utilizados para diferenciar objetos entre as imagens. No capítulo três mostramos como podemos fazer a ligação entre as imagens usando esses pontos. No capítulo quatro fazemos a sobreposição das diversas imagens segundo os pontos que obtivemos. No capítulo cinco fazemos uma filtragem para aproximar a imagem 3D um pouco mais da realidade. Por fim, no capítulo seis, analisamos os resultados obtidos e comentamos sobre os mesmos.

A metodologia utilizada para a elaboração deste projeto foi estar atento durante as aulas práticas e teóricas, enriquecidas com a pesquisa bibliográfica.

1 Processamento dos argumentos da função

No matlab decidiu-se usar ”cells” de uma forma a facilitar a alocação de memória que vai ser utilizada durante todo o funcionamento do nosso programa. Isto diminui o tempo de execução do programa e permite não ter nenhum ”warning” visto que este tipo de dados permite guardar qualquer coisa dentro dos mesmos.

O código divide-se apenas em duas funções, a ”reconstruct” e a ”ransac”. A primeira é a função principal. É nela que está todo o nosso programa, é esta função que permite fazer a reconstrução 3D. A segunda função é chamada dentro da anterior e permite-nos filtrar um pouco o resultado final de modo a aprocimá-lo mais à realidade como será explicado mais à frente no capítulo 5.

2 Recolha de ”Features” das imagens

Dadas as imagens RGB e respetivos valores de profundidade pela câmara kinect, queremos juntar as imagens todas no mesmo referencial de maneira a obter uma única imagem com toda a informação de todas as imagens junta. Para isso, necessitamos de poder reconhecer pontos em comum em várias imagens para poder juntá-los na imagem final. Ou seja, caso haja o mesmo objeto em duas imagens diferentes, este não pode aparecer duplicado na imagem final, porque no ”mundo real” apenas existia um objeto e é isso que a imagem final deve retratar. Isto não se consegue se considerarmos todos os pontos de todas as imagens, visto que podem haver vários pontos que sejam parecido mas que não

representem o mesmo ponto no mundo 3-D. Por exemplo, se houver uma parede branca no cenário retratado, vários pontos dessa parede numa das imagens são comparáveis a vários pontos dessa mesma parede noutra imagem, mesmo que não sejam os mesmos. Outro exemplo deste problema é o tabuleiro de xadrez, em que os cantos do quadrado são todos idênticos, o que poderia levar a juntar cantos diferentes ao juntar várias imagens, o que não é o desejado.

Para ultrapassar este problema, necessitamos de saber analisar as imagens de modo a retirar apenas pontos com informação essencial e que não seja ambígua, uma característica distinta do objeto (feature). Estes pontos são geralmente conhecidos como pontos chave ou pontos de interesse (keypoint features). Estes "keypoints" estão associados a um "descriptor" que é o que descreve as características dessa pequena porção de pixeis em torno da feature.

As classes de features mais utilizadas são, por exemplo, os cantos ("corners"), as "edges", a cor, a intensidade, entre outros aspetos da imagem, consoante o objetivo pretendido e as imagens a analisar.

Contudo, existem alguns problemas a ultrapassar como a mudança de intensidade de luz, de escala ou de perspetiva. Como será de esperar qualquer uma destas alterações na imagem irá fazer com que as features detetadas sejam alteradas. Ainda assim, existem alguns algoritmos mais robustos que são invariantes a algumas destas mudanças.

Um exemplo de um algoritmo de procura destas features em imagens é o "Harris corner detector", que foi abordado nas aulas, e que procura por cantos na imagem, analisando as variações do gradiente nas direções x e y da imagem. Quando estas variações são grandes nas duas direções no mesmo ponto existe grande probabilidade de estarmos perante um canto da imagem. Apesar de ser um algoritmo relativamente fácil de implementar este algoritmo não apresenta bons resultados quando as imagens sofrem algum tipo de alteração abordada acima.

Com isto, o algoritmo utilizado por nós foi o SIFT (Scale Invariant Feature Transform), devido a apresentar bons resultados, mesmo sendo pouco exigente computacionalmente, em imagens que sofreram algumas alterações fotométricas. Estas características fazem com que seja um algoritmo robusto, sendo bastante usado globalmente em processamento de imagem.

Para usar o SIFT no Matlab recorremos à toolbox do Vl Feat [3] que nos dá acesso a uma função do SIFT já implementada. Basta fornecer como argumento a imagem em escala de cinzentos com precisão single do Matlab, da seguinte forma:

```
[F,d] = vl_sift(single(rgb2gray(im{i})))
```

A função retorna `F` e `d`, sendo que cada coluna de `F` corresponde a uma feature da imagem e tem o formato `[X;Y;S;TH]`, onde `X,Y` é o centro fracionário da janela 16×16 em torno do keypoint detetado. `S` é a escala e `TH` é a orientação em radianos. Estes últimos dois parâmetros não são usados no nosso projeto.

Em cada coluna da matriz `d` está representado o descriptor da feature correspondente. Cada vector tem dimensão 128 e é da classe UINT8 do Matlab.

não deviam escrever em inglês joga contra!

Contents

é mais difícil e eu não falo!

1 Introduction

The main objective of the project described in the current report is the reconstructing a 3D scenario. To achieve it, RGB (colour) and depth images were used. It is made by fusing 3D cloud points.

It is assumed that the RGB and depth images were acquired simultaneously, so there is a temporal correspondence between the two images. Also, the 3D disposition between the two cameras is constant through all the data acquisition. This way, it is possible to map the 3D data from the depth image to the 2D RGB image.

The 3D reconstruction involves locating the camera in space. If we knew exactly where the camera was in each RGB and depth acquisition, then the solution would be straight forward. Knowing the camera pose in the moment of acquisition, it would be only necessary to project the depth image to a common 3D map. This is what will be done after estimating the pose. This will be presented in Section 1.1.

1.1 Problem Statement

Since there is no data on how the camera moved while acquiring the data, the only way to know its pose is using the RGB and depth images.

The depth image gives a (relatively) accurate description of the 3D world on its front. While it provides information of the physical structure, it neglects the colour and texture. On the the other hand, the RGB image does not provide good physical data but it evidences texture. The Figure 1 better explains the data acquired by both cameras. Combining the information from the two will help locating the camera.

(a)

(b)

Figure 1: On 1(a) is the RGB image and on 1(b) is the depth image. While the first conveys texture information, the other conveys the distance to object (or depth). The RGB image provides information on the posters and the signs on the door, the depth distinguishes minor distances between the door plane and the wall plane.

The acquired images are both noisy, so basing the basing the estimation in the minimum required data is not a possible approach. To minimise this error, should be used as much matches between images as possible.

This way, the pose estimation becomes an error minimisation problem, where the error to minimise is the distance between matches of the points clouds.

It was assumed that all images are fed to the algorithm in the same sequence as they were acquired. It also an assumption that the images have some degree of overlap with the previous and the next captured images. In the overlapping part are required enough texture and characteristics to identify by the feature extractor.

2 Implemented Solution

It is needed a mathematical model for the camera, this is presented in the Section 2.1. The models map 3D points to 2D coordinates in the image. They also map the camera 3D coordinate frame to other camera frames. The coordinate frame transformation can also be used to find the relative pose for the camera between two different acquisitions.

To minimise the image registry error the Iterative Closest Point (ICP) algorithm was used. The issue with this algorithm is the need of proximity between the initial points clouds. If they are not close enough, the outcome will be one of the multiple local minima. Therefore, the initial configuration should be as close as possible to a desired local minimum.

It is thus necessary to approximate them before applying the ICP. The implemented approach comes from 2D data progressively to 3D. First, there is the feature extraction with Scale Invariant Feature Transform (SIFT), described in Section 2.2. It is followed by Section 2.3, with the matching between SIFT descriptors of two images captured in two different instants.

The two images, depth and colour, are not captured in the same plane. There is an association problem of 3D point to pixel issue addressed on Section 2.4.

After associating 2D matches to 3D points it is possible to calculate an optimal rigid body transformation. The topic is addressed in Section 2.5

Since it is only being used local descriptors of 2D images, there might be a three-dimensional inconsistency between matched 3D points. This is exploited by the Random Sample Consensus (Ransac) in Section 2.6.

From the Ransac come a rotation matrix and translation vector as an initial approximation of the two points clouds. Only then is used the ICP described in Section 2.7.

There will be more than only two images for the 3D reconstruction. To this point, there is a global error minimisation problem. The ICP should minimise the global error since it finds the optimal register between points clouds. This topic is further discussed on Section 2.8.

2.1 Models

In this section the camera's intrinsic and extrinsic parameters are presented. The intrinsic parameters model the camera transformation of 3D points to 2D points, while the extrinsic model the camera pose in the world. The models are presented in projective coordinates, which linearises an otherwise non-linear problem.

The used approach is the pinhole model. It assumes that light coming from the seen object goes through an infinitely small hole (pinhole). Therefore, there is only one correspondence between a point in the 3D space to a point in 2D space. Then, we have

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & f_s & C_x \\ 0 & f_y & C_y \\ 0 & 0 & 1 \end{bmatrix}}_{K} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{1}$$

where *lambda* is the depth factor, u and v are the pixel coordinates, f_x and f_y are the pixel size factors, f_s is the skewness factor, C_x and C_y are a the centre of the pixel coordinate frame and X, Y and Z are the 3D point's coordinates. The matrix K is known as the camera matrix.

To map the depth camera to the RGB camera are used the extrinsic factors. These are a rigid body transformation described in Equation (2). R is a rotation matrix and T is a translation vector. The rotation matrix is an orthogonal matrix, which means it has determinant 1 and $RR^T = I$, or in another way, its inverse is given by the transpose $R^{-1} = R^T$.

$$\begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix} = R_{12} \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix} + T \tag{2}$$

Introduction

One of the most common problems in robotics is SLAM (Self Localization and Mapping), i.e., how to program a robot to define its own position in the world frame while mapping the environment around itself. We can describe this problem in a simpler way, by taking pictures of a given object with either a camera in different positions or several cameras placed around the object. In order to map each image correctly (since the camera positions are unknown at first) we need to define transformations between the points in the frames of each camera and the world frame we choose. However, defining these transformations is not a trivial task, because it is extremely hard to match points in different images.

In this project, we will develop a program that takes a set of 2D images and finds the correspondence between points in them, in order to define the transformation between their frames (rotation and translation). It is important to note that since we are using a 2D representation of 3D points we require additional information. In this case, this information corresponds to the depth of each pixel, which results in a (x,y,z) point. Consequently, it is easy to conclude that the previously mentioned camera is actually a set of two synchronized cameras (RGB and depth). This camera is called Kinect and it uses the time of flight of an IR ray to estimate the distance between the object and the camera, which corresponds to its depth in the 2D image. The second component of the camera is a regular RGB camera that captures the colour information for each 2D pixel. Lastly, since both components of the Kinect are at a fixed distance from each other, it's easy to match each RGB value to its correspondent depth value by means of triangulation.

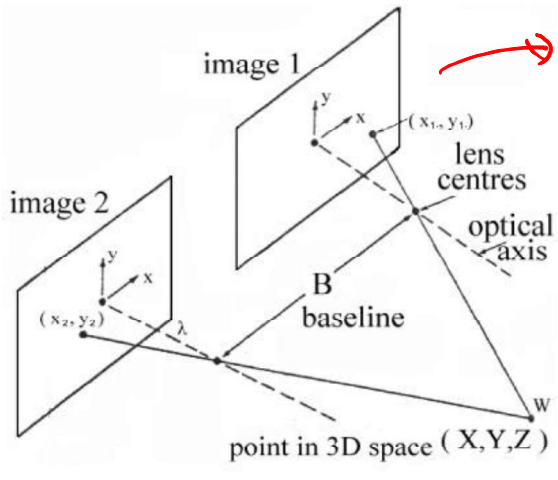


Figure 1

Solutions

In this section, we are going to present the methods and mathematical functions used to solve the problem of this project. We also used a *Matlab* toolbox called *vl_feat* to use some functions for image processing.

VL_Feat

The first function we used from this toolbox is called *vl_sift* and is used to identify important points (features) in images. The other one is *vl_ubcmatch* and tries to match corresponding features in different images.

Procrustes Method

Procrustes is a problem which consists of mapping points in A to B by means of an orthogonal matrix **R**, i.e.,

$$R = arg\ \min_{\Omega} \|\Omega A - B\|_F^2$$

where

$$\Omega^T \Omega = \mathrm{I}$$

and

$$\|\Omega A - B\|_F^2$$

represents the Frobenius Norm.

To solve this problem, the so-called Procrustes Method is commonly used. It consists in finding the nearest orthogonal matrix (**R**) to a given matrix

$$M = BA^T$$

using the SVD (Single Value Decomposition) matrix factorization, that is,

$$M = UDV^T$$

where

$$R = UV^T.$$

It should be noted that the matrices A and B contain, not all the image points, but the matched features previously obtained.

However, before this step, there is the need to superimpose the images. In order to do this, we calculate the centroids for all the images and subtract them with the points in each image. These subtractions correspond to centring all the point of each image around the origin of the world frame.

After having calculated the **R** matrices, the next step is to calculate the translation **T** between the images. To do that we simply subtract the centroid of the first image by the centroid of the second image rotated by the $\mathbf{R}_{21}$ (2 to 1), i.e.,

The reliability of our method for reconstructing 3D scenes is highly dependent on the number of correct matching pairs obtained with Sift. When the number of correctly matched points is high we get a resultant reconstruction that is very close to the real environment. On the other hand, when the number of correctly matched points is low, like with the dataset called *carSmall*, we can see that the point clouds are not merged correctly. This is not surprising as with a larger number of points, we should expect a more accurate estimate of the parameters.

If the percentage of inliers is very low it can be suspected that this set of points does not correspond to any true model. RANSAC will always return a set of inliers with at least as many points as it is needed to compute the parameters. Thus, after obtaining a final set of inliers, a way to improve one's result would be to only use a specific image if the number of inliers is sufficiently high.

To finalize, we concluded that the work developed for this project is a valid process for 3D environment reconstruction but it is still dependant on the quality of the images used as input, and it still needs to be perfected.

→ should have shown in the experiments!
I have to believe rather than evaluate by reason!

Depois da breve introdução de como as imagens RGB e depth são captadas, estão reunidas as condições para passar à explicação de como uma point cloud é obtida através de uma imagem depth, processo utilizado na realização deste projecto.

Existe uma relação matemática entre a localização de um determinado pixel, com a distância a que esse mesmo ponto no mundo real, se encontra da camara. Esta relação é válida para ambas as coordenadas, X e Y, sendo que a distância é a coordenada Z. Isto significa que conhecendo a coordenada u ou v de um pixel de uma imagem, e sabendo pela imagem depth, a distancia real Z desse ponto, é possível obter as coordenadas reais X ou Y. O primeiro passo para se obter uma point cloud, começa por corresponder a cada pixel de coordenadas u e v, a sua coordenada real X e Y, respectivamente.

Após este primeiro passo descrito anteriormente, já é possível obter uma point cloud, que, no entanto, não tem associado a cada ponto o respectivo código RGB, sendo que na imagem 3D obtida, são atribuídos aos diferentes planos, diferentes cores, começando no plano próximo de 0, com azul escuro, e acabando no plano mais distante com amarelo.

Embora com o resultado anterior já seja possível ter uma imagem 3D da cena captada, esta ainda não está reconstruída da forma pretendida porque para se ter uma noção exacta do que a camara captou, é necessário atribuir a cada ponto da point cloud, um código RGB.

Para se obter uma point cloud que seja uma representação bastante próxima da cena captada, é necessário converter as dimensões da matriz da imagem RGB, por forma a que sejam compatíveis com a matriz dos pontos nas 3 dimensões reais. Depois deste passo, é possível obter a point cloud desejada.

A base matemática que sustenta o que foi descrito anteriormente em relação às point clouds, é bastante simples e intuitivo. O ponto de partida é o modelo da camara, que matematicamente é descrito por

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} K_x & 0 & C_x \\ 0 & K_y & C_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Onde, u e v correspondem a coordenadas dos pixels e X, Y e Z correspondem às coordenadas reais. A matriz que se multiplica pelas coordenadas reais dos diferentes pontos corresponde à matriz com os parâmetros intrínsecos da camara.

Por outro lado, sabe-se que as coordenadas dos diferentes pixels resultam de,

$$u = f\frac{X}{Z} \tag{2}$$

e de

$$v = f\frac{Y}{Z} \tag{3}$$

em que no caso ideal, $f = 1$.

Fazendo alguma manipulação das expressões anteriores, e assumindo que são conhecidas as localizações dos diferentes pixels, chegam-se aos seguintes resultados

$$X = \frac{Z(u - C_x)}{K_x} \tag{4}$$

$$Y = \frac{Z(v - C_y)}{K_y} \tag{5}$$

que são as coordenadas reais, obtidas a partir das características intrínsecas da camara, da imagem depth, onde se encontram os diferentes Z's, e da localização dos diferentes pixels que constituem a imagem.

Com os pontos obtidos pelas expressões anteriores, e sabendo o código RGB correspondente a cada ponto são conhecidos todos os dados para se obter a point cloud da cena captada pela camara.

2.2 Similaridades entre imagens

2.2.1 Registo

É necessário obter pontos/zonas relevantes nas imagens. Para tal utiliza-se o algoritmo SIFT nas figuras RGB originais.

Visto que se obtém imensos pontos, é necessário fitrar os mais irrelevantes, ou com menor intensidade, utilizando um treshhold, em que só se obtém pontos superiores a este valor. Isto não só permite obter, mais à frente, uma melhor estimativa, como aumenta a rapidez do algoritmo.

2.2.2 Matching

De seguida, faz-se o matching de duas imagens consecutivas, ou seja determina-se pontos em comum em ambas. Utiliza-se, para isto, o algoritmo UBCMATCH, que pega nas descrições dos pontos dados pelo SIFT, das 2 imagens, e obtém pares de pontos semelhantes.

2.3 Determinação da transformação entre as duas imagens

Tendo-se os pontos em comum de ambas as imgens, pode-se então passar à determinação da sua transformação matemática.

2.3.1 Pontos com profundidade nula

Mas primeiramente, é necessário retirar os pontos com profundidade nula, ou seja, em que não foi possível obter profundidade. Isto pois, mais à frente, iria causar problemas pelo facto de não se ter as suas posições 3D, não sendo então possível usa-los para estimar a transformação das imagens.

Deste modo, retira-se não só as regiões da imagem RGB, como também os matches, com profundidade nula.

1 Introduction

This paper is written as part of a project of the course Image Processing and Vision. The project that this paper is addressing consist of a practical 3D reconstruction problem that is solved using Matlab. The problem concerns Self Localization And Mapping, also known as SLAM. This paper will introduce some theory that makes up the foundation for this problem, and thereafter go into details of how the problem has been solved.

The problem can be explained as follows. If you are to imagine a robot moving in an indoor scene, equipped with a Kinect sensor, the challenge is to build a 3D model of the scene as well as tracking the movements of the robot based on the information from the sensor. The Kinect camera includes a depth camera and a color camera and is thus providing the depth and color associated to each pixel in the photo. Since the Kinect sensor only provides a *local* representation of the scene, it is necessary to collect information from multiple images in order to achieve a complete representation. These images have to be aligned and fused into one single 3D model of the environment. For each image, the position and orientation of the Kinect has to be computed with respect to the scene coordinates. At last, the 3D cloud of points obtained at the current position needs to be fused from information obtained previously.

2 Theoretic background

In order to solve the problem presented in the introduction, we need to go into some theory. It will here be presented formulas and matlab features that can be useful when writing a function for solving the problem.

2.1 Formulas

The problem addressed in this project concerns 3D rigid body transformations. A 3D rigid body transformation is a mapping in 3 dimensional space that preserves both the distances between points as well as the orientation. The formal definition of a rigid transformation is a transformation that produces a transformed vector $\psi(\mathbf{X})$ when acting on any vector $\mathbf{X}$. In Cartesian coordinates in space, it can be expressed as in equation 1.[2]

$$\psi(\mathbf{X}) = R\mathbf{X} + \mathbf{T} \tag{1}$$

In the expression above, $\mathbf{T}$ is a translation vector and R is an 3×3 orthogonal matrix where $det(\mathrm{R})=1$. If we assume that there exist two corresponding point sets m(i) and d(i), for i = 1, ..., N, they are related according to formula 2

$$d(i) = \mathrm{R}m(i) + \mathbf{T} + \mathbf{V}(i) \tag{2}$$

In the equation, $\mathbf{V}(i)$ is a noise vector, while R and $\mathbf{T}$ are already defined. Solving for the optimal transformation [$\mathbf{R}$, $\mathbf{T}$] that maps the set $m(i)$ onto $d(i)$ normally requires minimizing a least squares error criterion. This criterion is presented in equation 3. [1]

$$\Sigma^2 = \sum_{i=1}^{N} \|d_i - \hat{\mathbf{R}}m_i - \hat{\mathbf{T}}\|^2 \tag{3}$$

Singular value decomposition (SVD) can be useful in this setting. SVD is a factorization of a real or complex matrix. Assume we have a m×n matrix called $\mathbf{M}$, and this matrix has entries that comes from the field K, that is either the field of real or complex numbers. Then we have a singular value decomposition of $\mathbf{M}$ described in equation 4. [3]

$$\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{4}$$

In this equation, U is a m×m unitary matrix and Σ is a diagonal matrix of size m×n with non-negative real numbers in the diagonal. V^* is conjugate transpose of the unitary n×n matrix V. From the equation, you can define the rotation matrix as R = V·U. The translation can be expressed based on equation 2 and 4 as T = mean(d)'-R·mean(m)'

2.2 Matlab features

A key component for solving this problem can be to use the Matlab toolbox VLfeat. VLfeat is an open source library of computer vision algorithms. It was created in 2007 by Andrea Vedaldi and Brian Fulkerson. One of the many algorithms it includes is the Scale-Invariant Feature Transform (SIFT), which was published by David Lowe in 1999. The SIFT algorithm can detect and describe local features in images. A feature, or keypoint, is a selected image region with a circular shape and an orientation. This algorithm can be used in many different settings, such as object recognition, image stitching, 3D modeling etc.

The function *vl_sift* can be used to get the detectors and the descriptors of a grey-scale image in single-precision. The detector extracts a number of frames or attributed regions from an image, and the descriptor is a 128-dimensional vector where each column corresponds to a column in the frame.

The descriptors are needed to use another function *vl_ubcmatch*, which matches the set of descriptors. The function will for each descriptor of an image1 find the closest descriptor in image2. It is a match if the distance between a descriptor d1 to descriptor d2 is significantly smaller than the distance from d1 to any other feature in image2. The function stores the index of the original match and the closest descriptor and also the distance between the descriptors.

To find the rotational matrix for the rigid body transformations the unitary matrices U and V are needed. They are found by using singular value decomposition, by the MATLAB-function *svd(H)*. This function corresponds to equation 4 in section 2.1.

The MATLAB-function *pointCloud* can be used to create objects for storing 3-D point clouds.

To get a rigid transformation that registers a moving point cloud to a fixed point cloud the function *pcregrigid* can be used. This function is based on the iterative closest point algorithm, which minimizes the difference between two clouds of points. The function also returns the transformed point cloud.

To avoid including points from matches that are not inliers, the RANSAC (Random Sample Consensus) algorithm can be implemented. It starts by selecting random feature pairs, and then determining the samples that are within an error tolerance of the generated model. These samples are considered as inliers, while the rest as outliers. If there is enough inliers, the model gets updated. The process is repeated for a number of iterations, and ends up giving the model with the smallest average error.

The function *get_xyz_asus* was given in lab classes. This function returns the xyz-coordinates in 3D-space from the UV-coordinates in 2D-space. Another useful function is *get_rgbd*. This function gets the color corresponding to each xyz coordinate.

3 Solution

The function that solves the given problem is named *reconstruction* and it takes images' names, intrinsic depth parameters of Kinect camera, intrinsic rgb parameters of the Kinect camera as well as rotation matrix and translation vector as inputs. The function returns *pcloud* and *transforms*. The *pcloud* is a N×6 matrix with the 3D point and rgb data of each point. The *transforms* is an array of structures with the same size as the input image names. It contains two elements: rotation matrix and translation vector, where they both contains the transformation between the depth camera reference frame and the world reference frame for each image.

Inside the *reconstruction* function, the *get_xyz_asus* is used to get the x, y and z coordinates based on the depth of each image. The output of this function is used to create a "virtual image" together with the rgb values, that again is used to create a point cloud. Furthermore, a function named *getMatchinPoints* is used to match points in two images. This function uses the toolbox *vl_feat*, described in section 2.2. The function uses *vl_sift* and *vl_ubcmatch* to do the matching. If there are less than 4 matched points, the function gives out an error.

When the images have been matched, the rgb point corresponding to each point in the depth image is collected and it is given depth. This is done using the function *get_xyz_asus* and thereafter a func-