SOROS: Sciadro Online Reconstruction by Odometry and Stereo-Matching

Fabio Ganovelli, Luigi Malomo, and Roberto Scopigno

ISTI-Visual Computing Laboratory

ABSTRACT

In this report we show how to interactively create 3D models for scenes seen by a common off-the-shelf smartphone. Our approach combines Visual Odometry with IMU sensors in order to achieve interactive 3D reconstruction of the scene as seen from the camera.

Introduction

The last decade has witnessed the release of a number of open source/commercial tools for 3D reconstruction from images. Bundler[1] and VisualSFM[2] have been followed by Agisoft Metashape (formerly Photoscan)[3], Autodesk ReCap[4] and Pix4D[5], are examples of tools that compute a possibly dense point cloud taken as input a set of images. Along with the sotware improvement, also low-budget hardware began to offer features that, prior to ten years ago, were only available on specialized high-profile devices. The most prominent example of this technology improvement is definitely on the smartphones, where high resolution cameras, inertial sensors an high-end processing units are packed in few centimeters. Concerning airborne 3D acquisition, we can find similar improvements. Instead of using a man-manned aero veichle carrying a heavvy laser scanner, we can now use remotely flown drones carrying range sensors within 2 kilos (please refer to[6] for a review). However, range sensors mountable on a drone are not yet a consumer device and their price range goes from few thousands o few tens of thousand euros. One difference between image-based and range camera acquisition is that the latest can provide a detailed description of the 3D environment in real-time, that is, the device returns a depth map of the acquired region instantly. This difference becomes very important if some form of unmanned control must be implemented, because an update 3D description of the environment allows the control algorithm to fly the drone safely and purposely. In the project Sciadro[?] we investigated the possibility of performing real-time acquisition from an RGB camera mounted on a drone.

As aforementioned, there are several solutions for computing a point cloud from a set of images, but non of them is real time and designed to be incremental (that is, they consider the whole set of images at once). In this report we report on the design of a system that we named Sciadro Online Reconstrutor by Odometry and Stereo-Matching (**SOROS** from now on). The report proceeds as follows: in Section 1 we give an overview of how the system is structured, show its main parts and the technology involved; in Section 2 we detail how the initial estimation of the camera position is done directly on board on the acquiring device; in Section 3 we show the core of the system, that is, the part devoted the online reconstruction and visualization. Results and discussion are reported in Section 4.

1 Our system at a glance

SOROS sotware consists of an Onboard Component (**OC** in the following), running on a smartphone, which can be mounted on a drone, and a Ground Component (**GC** in the following) running on a desktop PC (see Figure 1). The smartphone is the device in charge of the acquisition, through video camera combined with IMU sensors. The **OC** performs a quick data processing to find a sparse set of data points and the camera trajectory. A subset of frames of the acquired video along with the camera position and the point cloud is then sent to the **GC**. The **GC** consists of two concurrent activities: processing the incoming data received from the **OC** and visualizing the outcome.

2 Onboard computation of Camera positions

The main goal of the **OC** is to solve the egomotion problem, that is, to infer from the video the camera position and orientation that will be used in the **GC** along with the images. With the improvements of the off-the-shelf hardware technology witnessed in the recent years, it is now possible to use Visual Odometry algorithm directly on the smartphone. Even better, we can leverage available SDKs developed by the main manifacturers (that is, Google Inc **ARCore** for Android OS and Apple **ARKit** for iOS) that have been created for enabling the users to create Augmented Reality applications. Please note that, in order to

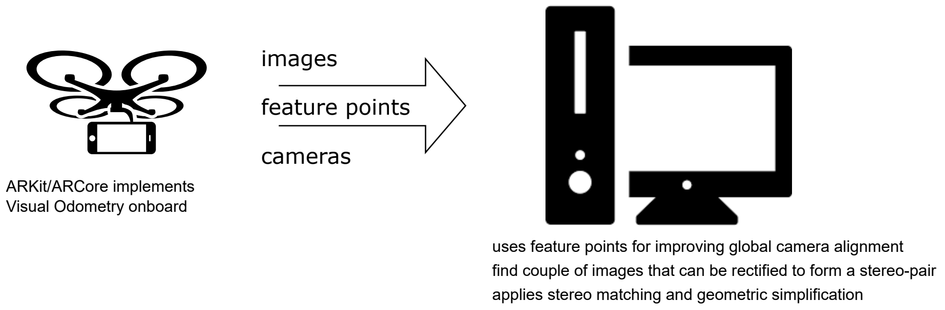
images
feature points
cameras
ARKit/ARCore implements Visual Odometry onboard
uses feature points for improving global camera alignment
find couple of images that can be rectified to form a stereo-pair
applies stereo matching and geometric simplification

Figure 1. Overview of the SOROS

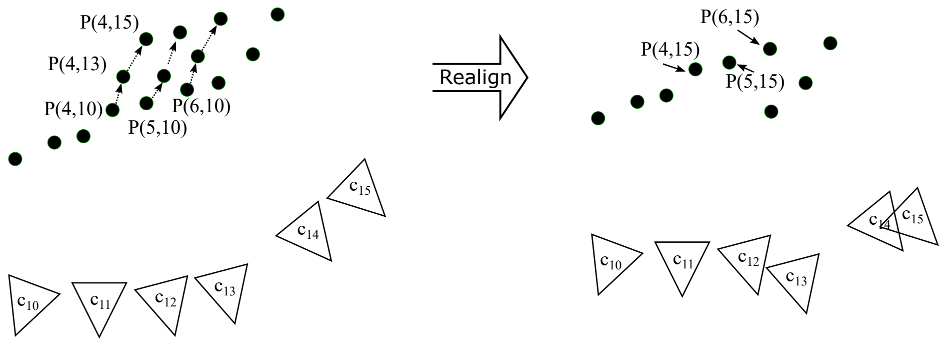

Figure 2. Left: cameras and features computed by the **OC**. Right: aligning cameras and features position in a common reference frame (see Section 3.3)

create a Augmented Reality application, we do not need a full 3D description the scene, we only have be able to maintain a consistent mapping between a few real 3D points of the world and their corresponding pixel position in the image. For example, in order to virtually place a vase on the center of a table, we only need to know where the centre of the table is in the image (or, more precisely, in the reference frame of the camera). Also, in order to know how big the virtual vase should be drawn, we need at least a second correspondence between reality and image. The position and orientation of the camera are not necessary to the purpose of creating an AR application but, luckily, they are a by-product of the computation and they are exposed by the ARKit SDK, as well as the list of computed 3D points. Therefore, we can use the ARKit SDK and obtain registered cameras and sparse point clouds, and of course a sampling of the video frames. More precisely, the output from ARKit consists of a sequence of triples

$$(I^i, M^i, F^i), i = 0 \ldots k$$

where: I_i is an image, M_i is the 4×4 world-to-camera transformation matrix and $F_i = f_j$ is a set of features. A feature corresponds to a physical point in 3D space which position is changed every few frames to keep the its projection on the 2D image consistent. If we specify with $P(j,h)$ the position p_j of feature f_j at time h we can write:

$$p_j^h = M_h \, P(j,h)$$

.

The scheme in Figure 2 shows a practical example. Feature 4 has been created at frame 10 in position $P(4,10)$. Then its position is updated at frame 13 and 15.

In an ideal scenario, the visual inertial odometry is error free, we obtain a single common reference frame and the 3D position of the point is computed once for all. Unfortunately, in practical cases the reference frame has to be updated. Also, we should keep in mind that AR applications are real-time and to find a globally consistent reference frame is not a requirement.

3 Online Asynchronous Recontruction with Stereo-Matching

The **GC** receives the sequence of triples transmitted by the **OC**. They already contain a point cloud that can be rendered but, as mentioned earlier, such point cloud is sparse and usually insufficient to have a meaningful visual feedback. Instead, we use stereo matching in order to create denser disparity/depth maps. The reason for using stereo matching is twofold. Firstly, the

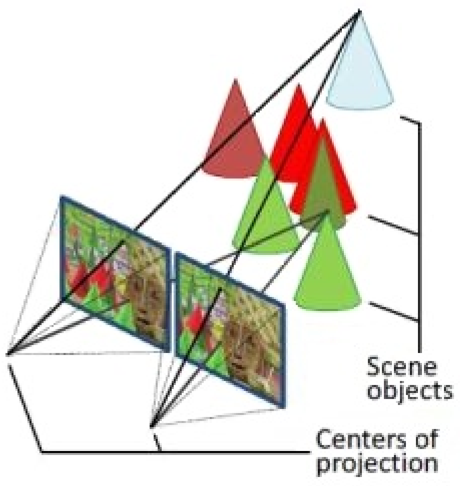

Figure 3. How stereo matching works. Knowing the relative position of the two cameras, the 3D position of any point identified in both images can computed by triangulation.

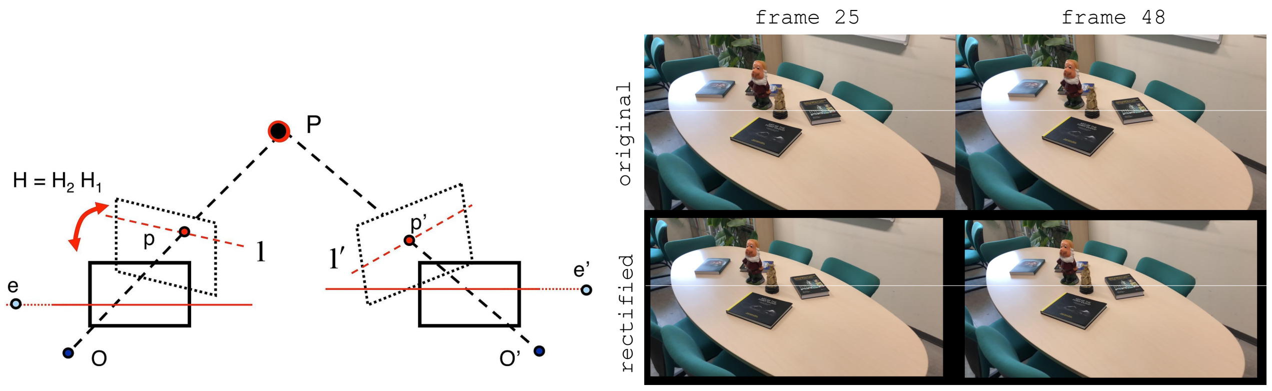

Figure 4. Given two cameras that do not share the same projection plane and are not separated by a horizontal translation, they can be rectified in order to comply to both requirements

camera movement is mainly translational, which implies that a large number of camera pairs can be easily rectified to create stereo-pairs; secondly the input consists of cameras which position is alrady known (because it has been computed by the **OC**).

3.1 Image pair stereo rectification

The stereo matching algorithm works by finding correspondences on two images taken by two registered cameras, that is, by two cameras whose relative position in known (see Figure 4). In order this process to be efficient, the two cameras are mounted in a rig so that their projetion plane is the same and that the corresponding pixels have the same vertical coordinates. In the context of Sciadro we do not have a stereo camera. Instead, we have a single camera moving in an essentially static scene. By design, the drone moves so that the mounted camera "scans" the scene by horizontal translation. Obviously, this does no guarantee that the camera will keep the very same projection plane between two close moments in time, but we can safely assume that the virtual pair (that is, the pair made by the same camera in two close instants) can be rectified so to form a stereo pair. *Stereo rectification* is the process of creating a stero pair from two cameras that do not share the same images plane and that are not separated by a horizontal translation. This is done by finding an average plane between those of the two cameras (See Figure 3.1), reprojecting the images onto such plane and redefining the camera intrinsic and extrinsic parameters accordingly[7].

The result of a stereo rectification consists of the new camera position and the warped images. Its quality is related to the amount of warping applied to the images. The more the two input cameras are near to be a stereo pair, the smaller the amount of warping necessary, the better the quality of rectification.

3.2 Finding the stereo pairs

Trying every possible pairing of cameras received from the **OC** in order to find suitable couples is obviously an inefficient strategy. If all the cameras where presented at once, we would uses an spatial data structure in order to be able to retrieve, for each camera, a small set of nearby cameras that can be rectified. Fortunately, the camera are given sequentially following the order in which the relative image were taken, therefore we can safely test each camera against the latest cameras to find suitable pairings.

Listing 1. Action upon reception of a new camera

```
map<camera,camera> allPC:
vector cameras;
OnNewCamera(c){
 Vec3f T; Mat3 R;
 cameras.push_back(c);
 int i=cameras.size()-2;
 do {
  Realign(c);
  T = Translation(c,cameras[i]);
  R = EulerAnglesRotation(c,cameras[i]);
   if( R_X < th_angle && R_y < th_angle && R_z < th_angle && cos(T,c.x))
    if(stereoRectify(c,cameras[i], rect_image_0, rect_image_1 ,Q)){
    stereoMatch(rect_image_0, rect_image_1, disp);
    allPC[c,cameras[i]] = createPointCloud(disp,Q);
    }
    i--;
 }
 while( (cameras.size()-i) < th_N && Norm(T) < th_distance );

}
```

Listing 1 shows a simplified c++-like code of operation are performed upon reception of a new camera from the **OC**. The do-while cycle iterates over the latest frames arrived until their distance is below a predefined threshold. For each one of these cameras (which are all permanently stored in the array `cameras`, we compute the difference between the two points of view (that is, the movement of the camera) and the rotation between the respective frames. If the rotation is below a given threshold and the translation if not too far from the x axis we process with stereo rectification of the cameras. Essentially, these are early discard tests that could also be omitted but please note that trying rectification involves some matrix inversion that we can avoid this way. If stereo rectification is successfull we compute the disparity map and hence the point cloud that is store in a `map` data structure, that is, a container that maps from pair of cameras to their computed point cloud.

3.3 Realigning the cameras

As explained in section 2, the cameras are not expressed in a single reference frame. This implies two things. Firstly, that the procedure just described may lead to unvalid point clouds because the can be compute by false stereo pairs. Secondly, that even correct point clouds are not globally aligned. To correct this misalignment we implement a `Realign` as follows. We define an affine transformation for each input camera c_i so that recomputing the camera transformations as $M'_i = A_i\, M_i$ express the cameras in a common reference frame. Initially A is set to the identity. Then, for every incoming frame, we check if there are at least 4 features which position has been updated. If there are, we define F_t as the affine transformation that maps the position of the updated features to their old position, otherwise we set F_t to the identity. Finally we update $A_t = F_t\, A_{t-1}$.

Referring to the example in Figure 2, let us assume that features from 1 to 6 are visible from camera 10 onwards. From camera 13, features 7,8,and 9 are added and position of features 4,5, and 6 is re-estimated to $P(4,13)$, $P(5,13)$, and $P(6,13)$ respectively. We found the best (in the least square sense) affine transformation F_{13} that maps $P(4|5|6,13)$ to $P(4|5|6,10)$ and compute $A_{13} = F_{13}A_{12}$ (in this example $A_{12} = A_{11} = A_{10}$. Then A_{13} is applied to camera c_{13} and all its associated features (that is from 4 to 9). Then, the position of features 4,5, and 6 are updatd again at frame c_{15} and the same process is repeated, bringing to the final configuration shown in Figure 2 (right).

4 Results and Future Work

We tested SOROS in a controller indoor scenario. The **OC** runs on a smartphone iPhone 8 while the **GC** runs on a desktop PC I7 3GHz Quad Core with 24 GB RAM. In this test setup the communication is simulated, that is, the **OC** write the output on a

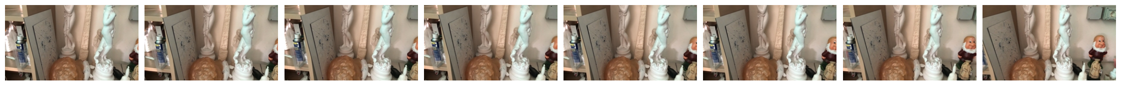

Figure 5. Few frame from the video acquisition.

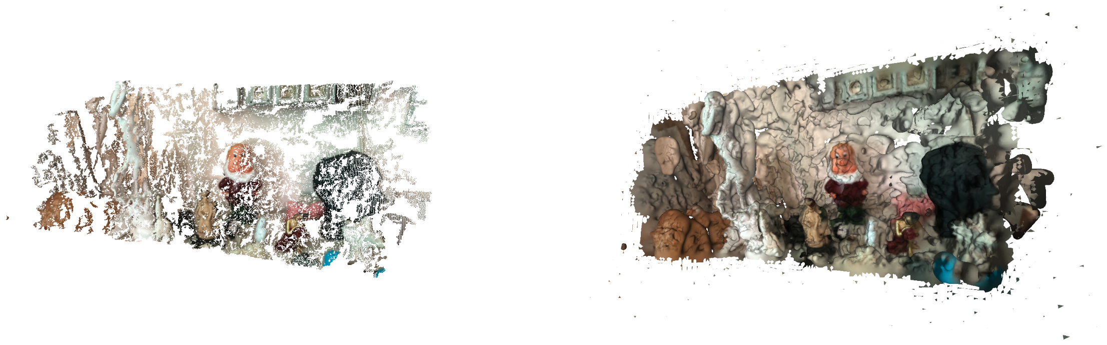

Figure 6. Result be runinng SfM on a subset of 50 images.

JSON text file, which is then transferred to the PC and loaded in a temporized manner, that is, as if it was sent directly from the phone. This is was done to allow us to test several solutions on the very same acquisition dataset.

As a first test, we exectuted a roughly translational movement that mimics the Sciadro setup in front of a table with few objects lying on top of it 5. In this setup, a large number of stereo pairs can be created. Typically each frame can be rectified togeter with several others, leading to a large number of stereo pairs. More specifically, the ARKit library provided 60 cameras per second in which our system can find over 200 stereo pairs/depth maps, an example of which is shown Figure 7. As a means of comparison, we run a structure from motion algorithm on a subsample of the data (50 images uniformly sampled over the entire timeline), densify the outputted point cloud (see Figure 6.left), perform a Poisson reconstruction and cleaned manually the find result, shown in Figure 6.right). As expected, the SfM-bsed reconstruction provides more accurate and detailed results. However, while SOROS is an online system providing an immediate 3D feedback of the environment, SfM reconstrution required over 3 minutes.

Although the early results are encouraging, there are still parts of the pipeline that needs further work before being deployed.

4.1 Accuracy

In the current implementation, the OpenCV[8] library is used both for rectification and stereo matching. OpenCV provides a few consolidated methods for implementing stereo matching but more recent and effective soluitons have been published for the specific of ARKit that wold be worth integrating. More importantly, at the present there are only a few heuristics for assessing the quality of each disparity map. As a consequence, the scene can rapidly become cluttered with outliers. Furthermore, the sparse point cloud provided by ARkit is only used in the realignment phase but not as a form of constraint to apply to the depth map generation.

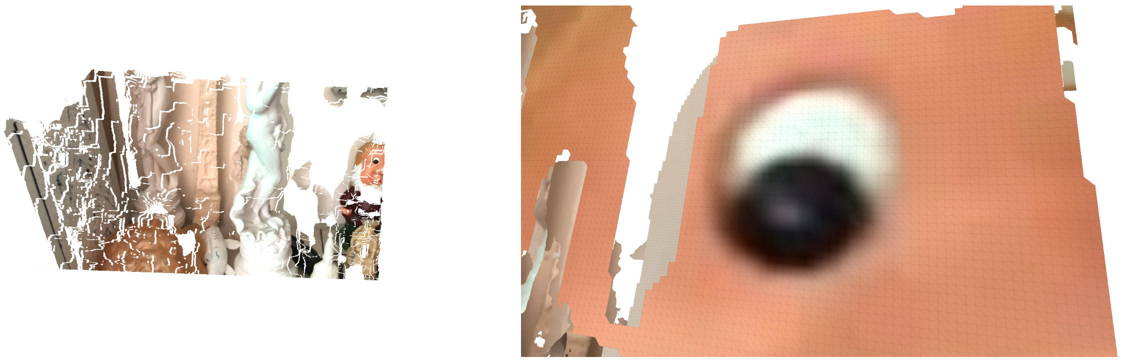

Figure 7. Results from online stereo matching. Left) rendering of a triangulated depth map Right) detail that show the tessellation density.

4.2 Data handling and Posst Processing

At the present, the computed depth maps are stored in main memory and optionally saved to disk, but a proper managment of the ever growing amount of data that would be produced in a longer acquisition time has not yet implemented.

As it can be seen in Figure 7.right, the tessellation obtained by connecting adjacent points of the depth maps is way too dense with respect to the accuracy of the point cloud and it significantly contributes to the memory overhead. This can be amended by implementing a post processing step where a mesh simplification algorithm is implemented in order to reduce the triangle count and, consequently, the memory footprint of the depth maps[9]. Also, overlapping depth maps could be finely aligned and merged[10] in order to provide an overall consistent 3D representation of the scene. Please that all the steps described can be quite computationally expensive and would require to be executed concurrently with the rendering.

Acknowledgments

This work was supported by the project SCIADRO, 2016-2019 Regione Toscana PAR FAS 2007-2013.

References

1. Snavely, N., Seitz, S. M. & Szeliski, R. Photo tourism: Exploring photo collections in 3d. *ACM Trans. Graph.* **25**, 835–846, DOI: 10.1145/1141911.1141964 (2006).

2. Wu, C. VisualSFM: A Visual Structure from Motion System. http://ccwu.me/vsfm/ (2019).

3. Agisoft. Agisoft Metashape. http://www.agisoft.com (2019).

4. Autodesk. ReCap. https://www.autodesk.com/products/recap/overview (2019).

5. Pix4D. Pix4D Mapper. https://www.pix4d.com/product/pix4dmapper-photogrammetry-software (2019).

6. Maria Antonietta Pascali, F. G. Deliverable D3.1: Requisiti e specifiche funzionali del sistema di acquisizione ed elaborazione dati. http://www.sciadro.it/ (2019).

7. Hartley, R. & Zisserman, A. *Multiple View Geometry in Computer Vision* (Cambridge University Press, New York, NY, USA, 2003), 2 edn.

8. Bradski, G. The OpenCV Library. *Dr. Dobb's J. Softw. Tools* (2000).

9. Garland, M. & Heckbert, P. S. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, 209–216, DOI: 10.1145/258734.258849 (ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997).

10. Schertler, N. *et al.* Field-aligned online surface reconstruction. *ACM Trans. on Graph. - Siggraph 2017* **36**, 77 (2017).