STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXI**, Number 1, 2016

PARALLEL TRACKING AND MAPPING WITH SURFACE DETECTION FOR AUGMENTED REALITY

ALEXANDRU MORARU, ADRIAN STERCA, AND RAREŞ BOIAN

Abstract. This work presents PTAM-SD, a Parallel Tracking and Mapping algorithm with planar surface detection suitable for augmented reality applications in small workspaces. Our PTAM-SD algorithm tracks the movements of the video input camera in 3D space and updates the scale and position of the augmented reality game scene. PTAM-SD also assists the AR game by determining a planar surface in the real environment which is used for displaying the game scene.

1. Introduction and Related Work

Simultaneous Localization and Mapping (SLAM) is the process through which a mobile robot builds a map of the surrounding environment and at the same time it tracks its movement through this environment map [1]. The robot uses odometry data to estimate its position and sensor data to extract landmarks from the environment, landmarks which will form the map of the environment. Examples of sensors which can be used for landmark extraction and mapping are laser scanners, sonars and, more recently, video cameras. Formulated mathematically, the SLAM problem implies computing the probability distribution $P(X_k, m|Z_{0:k}, X_{0:k-1})$ for each time k where X_k is the location of the robot at time k, m is the map (i.e. the set of locations of all landmarks observed so far), $Z_{0:k}$ is the set of all landmark observations in time interval $[0, k]$ and $X_{0:k-1}$ is the set estimated locations of the robot in time interval $[0, k-1]$. The algorithm for solving the SLAM problem usually has an iterative form and at each step the new position of the robot is estimated based on a motion model, then observations are gathered from the sensors and

Received by the editors: December 31, 2015.

2010 *Mathematics Subject Classification.* 68T45, 68U10.

1998 *CR Categories and Descriptors.* I.4.8 [**Image Processing and Computer Vision**]: Scene Analisys – *Tracking, Surface Fitting*; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems – *Artificial, augmented, and virtual realities*.

Key words and phrases. parallel tracking and mapping, surface detection, augmented reality.

the new position is updated. Many SLAM solving algorithms use Extended Kalman Filters [2] or particle filters [4].

Several SLAM algorithms were proposed for augmented reality applications [3]. These algorithms use a hand-held camera or the video camera of a smart phone for getting sensor data from the environment. PTAM [3] tracks the movements of the video camera and builds a map of the environment in order to render an AR game. The localization and mapping processes are done approximately separately in two threads. It does not require a fiducial image for placing the game scene, but relies on the user to point the camera initially at a flat (planar) surface.

In this paper we propose a parallelized SLAM algorithm with surface detection used for building augmented reality applications for small workspaces. The augmented reality is projected on a planar surface of the environment and, differently from PTAM [3], the planar surface is automatically detected in the video input stream by the system.

An approach to finding planar surface for real time applications is by using segmentation. Segmentation is an important step in many perception tasks, such as object detection and recognition. In [6] an efficient method for segmenting organized point cloud data is presented. This approach uses the RGB data that are available in an image and on top of that uses the 3D coordinates of each pixel and of surface normals. There are two main algorithms that are proposed in this paper: Connected Component algorithm and a Planar Segmentation algorithm [6]. The Connected Component Algorithm operates on organized point cloud data. It works by partitioning an organized point cloud into a set of segments. The Planar Segmentation Algorithm segments the scene to detect large connected components corresponding to planar surfaces, including walls, the ground, tables, etc. Our approach of detecting planar surfaces involves building an approximate disparity/depth map by simulating 3D vision from consecutive frames with a slight OX displacement recorded by the same video camera.

2. System Architecture

The system uses a smart-phone (which will be referred to as the client) for getting sensor data from the environment (i.e. video input and accelerometer data) and for rendering the AR game, and a computer/notebook (which will be referred to as the server) for processing the sensor data received from the client (i.e. tracking the smart-phone position, mapping and planar surface detection). The client component has two running threads. On one of them, frames are captured from the video camera and send to the server and the other thread waits for incoming positioning data from the server; this positioning

data tells the client application the new, updated location of the video camera (relative to the previous location). The server component has similarly, a receiving thread and a processing thread. The receiving thread receives frames from the client component and pushes them in a queue, while the processing thread removes frames from this queue, computes/updates the current location of the video camera and the map and then it sends this updated location to the client. Together with this information, the server also sends to the client the position and contour of the planar surface it detected in the input video (if any).

The client component was implemented as a hybrid project in C# and C++ whilst the server application is entirely written in C++. All the communication between the server and the client is done using reliable, TCP connections via an ad-hoc wireless connection. The system was tested using a laptop with an Intel Core i5-2430M processor, with maximum processing speed of 2.5GHz and 6GB RAM, and a Microsoft Lumia 640 with an Quad-core 1.2 GHz Cortex-A7 processor, 1GB RAM and the GPU is Adreno 305.

3. Tracking and Mapping

The tracking of camera movements is based on a feature detection and matching algorithm and our system uses the support of the OpenCV 2.4.10 library for this. The tracking algorithm requires two consecutive frames received from a video input device, from which it detects and extracts ORB keypoints (i.e. a combination of FAST keypoint detector and BRIEF descriptor with some modifications) [12]. We have tested several feature extraction algorithms: SIFT [7], SURF [8], STAR (a version of [9]), MSER [10], BRISK [11] and ORB, and the one that gave the most accurate homography was ORB. Details about these tests are presented in [5]. In Fig. 1 we can see that ORB gives an accurate homography and a good matching between keypoints from two consecutive video frames with a small displacement on the 0x axis.

After the ORB feature extraction, the algorithm uses a BruteForce Matcher to find similar keypoints in both frames and from the resulting array of matching keypoints, the algorithm computes the minimum and maximum distance between two matching keypoints. Having these two distance values and a (minimum) threshold of 3 times the minimum distance for two keypoints, a series of "good" matches is extracted. After the filtering of keypoints, the algorithm computes the displacement (on x-axis and on y-axis) between the two images. This computation is done firstly by calculating the difference between every two keypoints. Secondly, we compute the average displacement between all the filtered keypoints. The homography is computed from the sets of keypoints (one set for each frame). The homography can be computed only

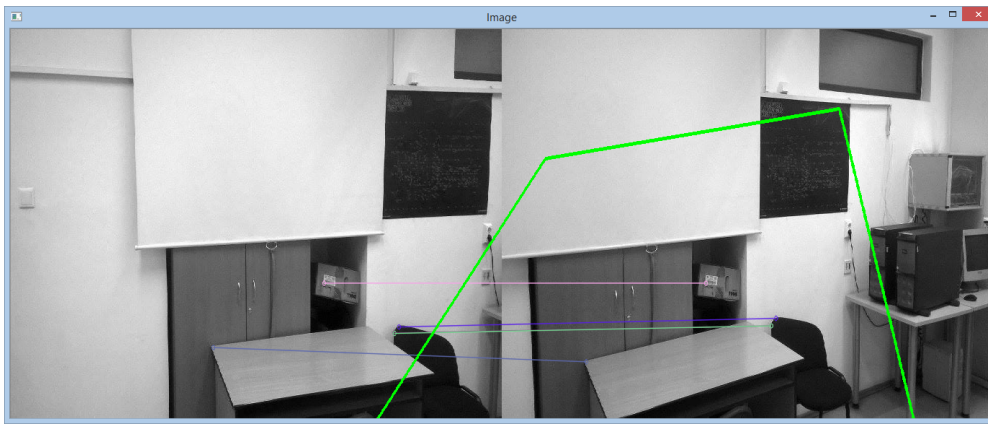

FIGURE 1. The matched keypoints and homography computed using ORB features for two successive frames with a small X displacement

if the size of both sets of keypoints is bigger than 4. The homography returns as result a matrix and from that matrix we perform a transformation of perspective between the two images. This perspective transformation gives a quadrilateral, which is the surface of the first frame transposed into the second one. The tracking algorithm is depicted in the following listing:

```
function FindDisplacement(image1, image2) {
        keypointsImage1 = detect(image1);
        keypointsImage2 = detect(image2);
        descriptorsImage1 = extract(image1, keypointsImage1);
        descriptorsImage2 = extract(image2, keypointsImage2);
        matches = BFMatcher.match(descriptorsImage1, descriptorsImage1);

        FindMinAndMaxDistances(matches, minDist, maxDist);
        goodMatches = DetermineGoodMatches(matches, minDist);

        <image1Points, image2Points> = GetImagesPoints(goodMatches);
        <displacementX, displacementY> =
                ComputeDisplacement(image1Points, image2Points);
        h = findHomography(image1Points, image2Points);

        perspectiveTransform(frame1_corners, frame2_corners, h);

        quadrilateralSurface = ComputeSurface(frame2_corners);
}
```

where *image*1 and *image*2 are the two images that are compared to find the displacement, *displacementX* is the computed displacement on the x-axis,

$displacementY$ is the computed displacement on the y-axis, h is the computed homography, stored as a matrix. The variable $quadrilateralSurface$ represents the value of the area of the first image transposed into the second one.

In order to compute the displacement on the z-axis (i.e. zoom in/out) we compute the surface of the quadrilateral resulted from transposing the first image into the second one (obtained by the algorithm depicted above). The surface of the quadrilateral is computed by drawing one diagonal of the quadrilateral and computing the surface area of the two resulting triangles using Heron's formula for calculating surface areas:

$$Surface = \sqrt{s * (s - a) * (s - b) * (s - c)}, \tag{1}$$

where s is the semi-perimeter of the triangle, and a, b, c are the length of its edges. The transposed image surface will be used to compute the displacement on the z-axis and to scale the elements from the virtual reality on the smartphone client.

4. Mapping

Simply put, mapping is the process of initializing and adding feature points in a 3D environment map while performing the tracking operation. The map is initialized with the keyframes from the first frame received. After that, whenever a new frame is received the mapping thread performs a search to find if the frame contains new key features that were not previously added into the map. In our system we use mapping for storing the representation and the spatial coordinates of the detected planar surface. In the map, the system saves the approximated coordinates and dimensions of the planar surface and its surroundings, such that when the video input device reaches in the neighborhood of the surface, the server will tell the client to draw the game characters on the planar surface.

The idea is to store some key features from the frame that contains the planar surface, and at every frame processing where we determine the displacement we shall also compute the relative displacement from the features that describe the planar surface. By making these repeated computation we can determine (with a small error) when we returned to the initial position.

5. Surface Detection

The surface detection algorithm tries to detect a planar surface area in the video input stream. It does this by computing a disparity map (i.e. 3D depth map) from each pair of consecutive video frames and selects from this disparity

map the largest area having the same (maximal) distance from the viewer. The algorithm requires two frames received from any input device with one video lens. The restriction imposed by this algorithm is that the two frames must be taken consecutively and the second frame must be displaced only on the x-axis (horizontally). In the disparity map the white pixels represents the closest object to the camera and the black pixels the most distant object. The algorithm's goal is to find a surface of continuous black pixels of a maximum area.

Due to the fact that in our PTAM-SD project we continuously send frames to the server (from the client), we can use two frames received. We do not take strictly two successive frames, we take a frame, skip a few frames (i.e. 5 frames) to reduce the computational costs and also to obtain an accurate disparity map, then take the next one and compute the disparity map. So the two selected, consecutive frames are 5-frames apart. Below, we present an example of the disparity map obtained from two images taken with our phone camera, that is also used to run the client in the main system. In Figure 2 we have a random frame saved from the video camera output stream. We then saved another frame from the same video stream which is 5-frames apart from the frame depicted in Figure 2. This second (i.e. which was obtained by slightly moving the video camera on the right on the 0X axis) is depicted in Figure 3. The depth map computed from these two frames is shown in Figure 4.

The resulting disparity map is not extremely accurate, but it is something we can work with. It contains the main objects in the scene which are represented as groups of lighter pixels and this is very helpful in the process of finding a planar surface. The drawbacks of this algorithm of finding a planar surface in a scene with multiple objects are:

- The human error; because a human cannot move the video input device just on the x-axis as wanted, unwillingly it will move it either on the y- or z-axis or even both, even if it is just a little movement.
- This algorithm alone cannot make the difference between a planar surface (like a table) and a wall.

The planar surface detection algorithm performs a search on the computed 8-bit disparity map. This algorithm searches in the received disparity map for the largest continuing area of black pixels. Because the computational time is important to us, we did not consider parsing all the pixels in the image, because this will mean a very large (greater than 100 milliseconds) processing time. Instead, we take a width step of 5 pixels and a height step of 10 pixels. So, we parse the matrix on rows in search for the longest sequence of black pixels. Having the array of longest sequence of black pixels from each row, we

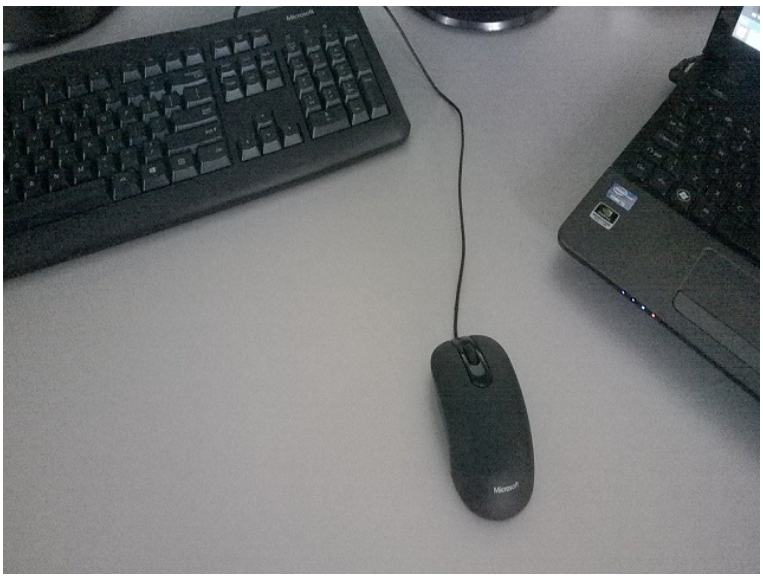

FIGURE 2. The initial frame

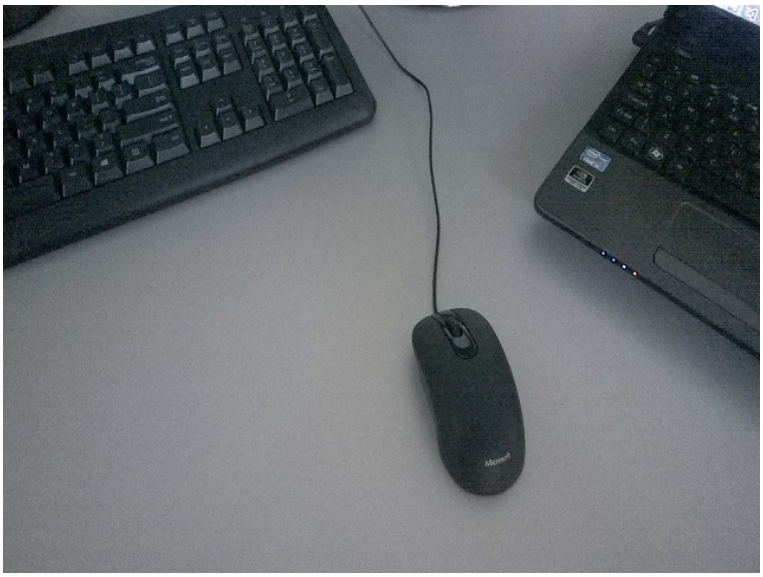

FIGURE 3. The displaced frame (obtained by moving the camera on the right)

FIGURE 4. The disparity map of the two images in Fig. 2 and Fig. 3

parse it in the search of the longest run of consecutive sequences that overlap and the length of the overlap part is greater than a minimum threshold. The result of all these computations are the coordinates (in pixels) of the planar surface found by the algorithm.

The full algorithm is available below, in pseudo-code:

```
function FindSurface(image)
{
     heightStep = 10;
     widthStep = 5;
     matrixLength = image.GetNumberRows() / heightStep;
     surfaceMatrix[matrixLength][2];
     for i = heightStep to image.GetNumberRows(), i = i+heightStep
     {
          for j = widthStep to image.GetNumberCols(), j = j+widthStep
          {
               color = image.GetColorAt(i, j);
               if (color == 'black')
               {
                    increase the current_sequence;
                    continue searching;
               }
               else
               {
```

```
                    end current_sequence;
                    if (current_sequence > max_sequence)
                    {
                         max_sequence = current_sequence;
                    }
               }
          }
          matrixPosition = i / heightStep - 1;
          surfaceMatrix[matrixPosition][0]=maxSequence.GetStartIndex();
          surfaceMatrix[matrixPosition][1]=maxSequence.GetLength();
     }
     return OperateOnSurfaceMatrix(surfaceMatrix, matrixLength,
                                   widthStep, heightStep);
}
```

where the parameter *image* is the disparity map with 8-bit pixels, *widthStep* and *heightStep* are the imposed steps for parsing the disparity map, and $surfaceMatrix$ is a two-dimensional array that keeps the coordinates of pixels (starting pixel and the number of pixels from the longest sequence of black pixels) from each visited line. The $OperateOnSurfaceMatrix$ function computes the largest rectangular surface enclosed in a black-colored pixels area of the disparity map.

```
function OperateOnSurfaceMatrix(surfaceMatrix, matrixLength,
                                widthStep, heightStep)
{
     for i = 0 to matrixLength - 1
     {
          for j = i + 1 to matrixLength
          {
               linePixels = ComputeOverlappingSurface(surfaceMatrix,
                                                      i, j, widthstep);
               if (linePixels is valid)
               {
                    int currentSurface = (linePixels.GetEndingPixel() -
                        linePixels.GetStartingPixel()) * ((j - i + 1) *
                        heightStep);

                    if (currentSurface > maxSurface)
                    {
                         save current surface size and its coordinates;
                    }
               }
          }
     }
     return surface_coordinates;
}
```

where $surfaceMatrix$ is the matrix containing the data regarding longest lines of black pixels, and *linePixels* represents the coordinates of overlapping segments of the lines, returned by function $ComputeOverlappingSurface$.

Variables $currentSurface$ and $maxSurface$ represent the area of the current rectangular surface candidate and the final chosen rectangular surface, respectively.

```
function ComputeOverlappingSurface(surfaceMatrix, position1,
                                   position2, widthStep)
{
     startingPixel = surfaceMatrix[position1][0];
     endingPixel=startingPixel+surfaceMatrix[position1][1]* widthStep;

     lineOverlappingThreshold = 50 * widthStep;
     for i = position1 + 1 to position2 + 1
     {
          lineStartingPixel = surfaceMatrix[i][0];
          lineEndingPixel=lineStartingPixel+surfaceMatrix[i][1]*widthStep;

          if (lineStartingPixel > startingPixel)
               startingPixel = lineStartingPixel;

          if (lineEndingPixel < endingPixel)
               endingPixel = lineEndingPixel;

          if (endingPixel - startingPixel < lineOverlappingThreshold)
               return invalid;
     }

     return <startingPixel, endingPixel>;
}
```

where the variable $lineOverlappingThreshold$ represents the minimum threshold distance that a mutual segment of the lines must meet in order for it to be considered valid.

Applying the search algorithm on the disparity map in Fig. 4, we obtain the planar surface highlighted in Fig. 5.

6. Virtual Reality Rendering

In order to evaluate our proposed PTAM-SD system, we developed a small DirectX based game for smart-phones running Windows Phone 8 or better. Our game is rendered on the smart-phone and uses the functions of PTAM-SD in order to track the phone's movement, render and scale the scene according to the phone's new position and detect a planar surface for mapping the game's scene on the found surface. We used several smart phones as our client besides the one outlined in Section 2. The frame rate at which the application renders the game scene is around 30 to 60 fps (frames per second), depending on the device running the application.

The game draws a main cube which the user can control by moving it on the X, Y and Z axis using buttons displayed on the phone's touchscreen. The

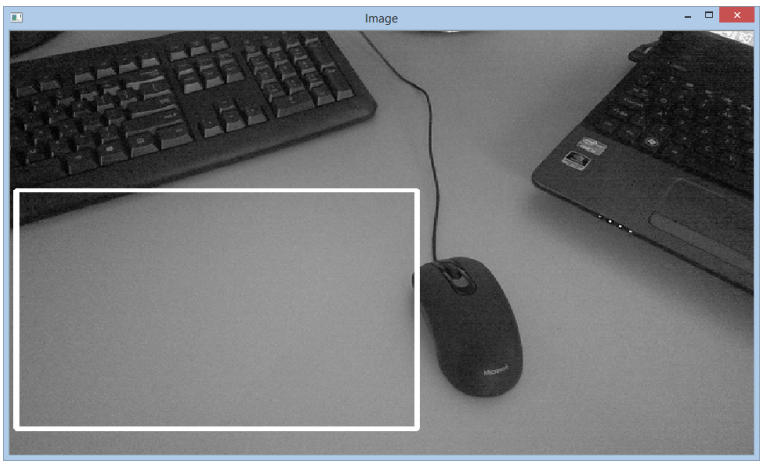

FIGURE 5. The planar surface detected

user must move the main cube so that it avoids collision with other cubes that are continuously passing horizontally (i.e. from left to right) below, on the same plane and above the main cube. Th user can also navigate in the AR scene by moving the phone in 3D space. PTAM-SD will detect if the phone moved and will adjust the game scene accordingly. PTAM-SD will also determine a planar surface on which the game would be rendered. We can see 2 captions with the game in figures 6 and 7. But we can not show the whole game here, so we recorded two videos with the game which are available here: https://www.cs.ubbcluj.ro/~forest/research/papers/augmentedreality-capture1.m2ts and https://www.cs.ubbcluj.ro/~forest/research/papers/augmentedreality-capture2.m2ts.

7. CONCLUSIONS

In this paper we presented PTAM-SD, a SLAM-like system that tracks the camera movement on the x-, y- and z-axis. This system also offers support for the cameras tilt by updating all the rendered scene according to its tilt angle and provides an important feature to AR applications that use our PTAM-SD system, namely a planar surface detection algorithm that does not require the user intervention, nor a fiducial image. The detected planar surface can be used to render an AR game on, providing the consumer with an improved user experience.

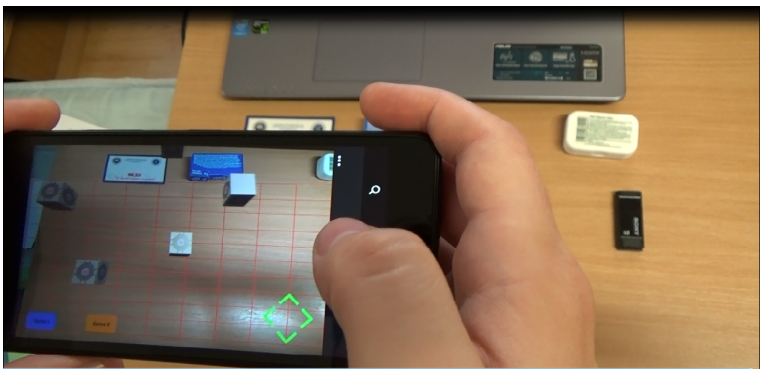

FIGURE 6. Game capture #1

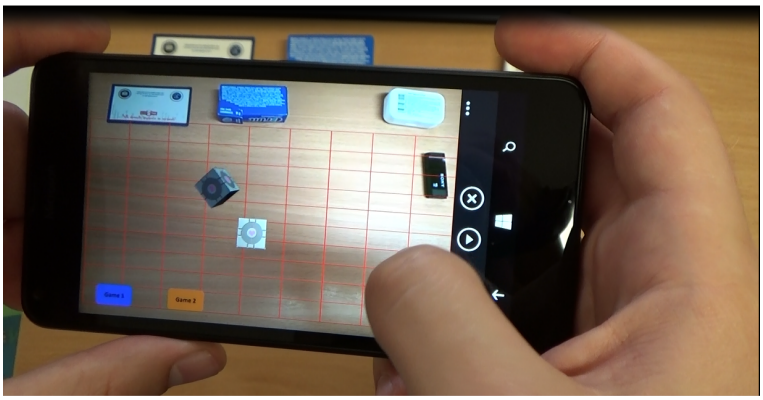

FIGURE 7. Game capture #2

REFERENCES

[1] H. Durrant-Whyte and T. Bailey, *Simultaneous Localization and Mapping: Part I The Essential Algorithms*, IEEE Robotics & Automation Magazine 13 (2), 99-110, 2006.
[2] H. Durrant-Whyte and T. Bailey, *Simultaneous Localization and Mapping(SLAM): Part II State of the Art*, IEEE Robotics & Automation Magazine 13 (3), 108-117, 2006.
[3] G. Klein and D. Murray, *Parallel Tracking and Mapping for Small AR Workspaces*, In Proc. International Symposium on Mixed and Augmented Reality, 2007.
[4] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, *FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges*, In Proc. International Joint Conference on Artificial Intelligence, pages 11511156, 2003.

[5] A. Moraru, *Parallel Tracking and Mapping with Surface Detection for Augmented Reality*, Diploma Thesis, Babes-Bolyai University, Dept. of Computer Science, July 2015.
[6] A. J. B. Trevor, S. Gedikli, R. B. Rusu, H. I. Christensen. *Efficient Organized Point Cloud Segmentation with Connected Components*, In Semantic Perception Mapping and Exploration, 2013.
[7] D. G. Lowe, *Object recognition from local scale-invariant features*, In Proceedings of the International Conference on Computer Vision 1999, pp. 11501157.
[8] H. Bay, A. Ess, T. Tuytelaars, L. Van Gool, *SURF: Speeded Up Robust Features*, Computer Vision and Image Understanding (CVIU), Vol. 110, No. 3, pp. 346359, 2008.
[9] M. Agrawal, K. Konolige, M. R. Blas, *CenSurE: Center Surround Extremas for Realtime Feature Detection and Matching*, In Proceedings of European Conference on Computer Vision 2008.
[10] J. Matas, O. Chum, M. Urban, and T. Pajdla, *Robust wide baseline stereo from maximally stable extremal regions*, In Proceedings of British Machine Vision Conference, pages 384-396, 2002.
[11] S. Leutenegger, M. Chli and R. Y. Siegwart, *BRISK: Binary Robust Invariant Scalable Keypoints*, In Proceedings of International Conference on Computer Vision 2011.
[12] E. Rublee, V. Rabaud, K. Konolige, G. Bradski, *ORB: an efficient alternative to SIFT or SURF*, In Proceedings of International Conference on Computer Vision 2011, pp. 2564-2571, 2011.

Department of Computer Science, Babeş-Bolyai University, Cluj-Napoca
E-mail address: maie1437@scs.ubbcluj.ro
E-mail address: forest@cs.ubbcluj.ro
E-mail address: rares@cs.ubbcluj.ro