Mobile control interface for modular robots

Luc Girod
Student: Luc Girod, Supervisor: Stéphane Bonardi, Massimo Vespignani, Professor: Auke Ijspeert
June 6, 2014

Abstract

The goal of this project was to create an augmented reality (AR) interface for modular robots called Roombots. Currently a 3D simulation of Roombots exists but it doesn’t use the capability of augmented reality to improve the user experience. We envision an application that allow the user to watch the simulated Roombots superimpressed to the real view from the camera, from any points of view by just moving in the environment as if there were real Roombots in the room. We considered many possibilities and we chose to use a SLAM-like algorithm because this doesn’t rely on any sensors or external beacons except the camera to works. I first begin to implement such tool and finally decided to modify an augmented reality software from Oxford called PTAM (Parallel Tracking and Mapping) to match our needs.

1 Introduction

In this project we would like to create an application allowing to display an augmented reality (AR) simulation of modular robots called Roombots in their real environment. Roombots are small modular robots which can attached themself together like building blocks to create self-assemble and self-reconfigurable furnitures. The solution should be able to runs this simulation as an augmented scene which stay in place regardless of the user position and angle of view. There are many already developed solutions, but we choose a system which doesn’t need extra devices or previous knowledge of the environment. I first begin to create such system on my own, but we found an open source program that we can modify to meets our requirements.

Figure 1: This is two Roombots connected together. (http://biorob.epfl.ch/roombots)

Figure 2: This is a 3D scene of Roombot with the non AR application.

2 Considered solutions

Among the considered solutions, the first was to recognize an already known pattern which indicates our orientation and, by knowing the camera projection, the distance to the pattern can be known. It’s also possible to get this distance with a depth sensor such as a Kinect. The problem of this method is the need of a pattern which must be in the field of view. Using sensors on the camera was an idea but the error of the position and orientation increases with each movement of the user. We decided to use a SLAM-like (SLAM stands for simultaneous localization and mapping) algorithm. It’s a set of

---

1http://biorob.epfl.ch/

2A depth sensor is a device that give informations about the distances of each object in the view


techniques to estimate the localization and to reconstruct the environment in 3D while the camera is moving. The advantage of such algorithm is that it creates itself a mapping of the room without markers. It doesn’t need any external sensors and no previous knowledge of the environment is needed. While developing an augmented reality API, we found a good solution from Oxford university called PTAM for Parallel tracking and mapping) which I tested and modified to match our needs.

3 First work

3.1 Features detection

In all SLAM algorithms we need to detect points of interest in the image. Detecting points of interest is a kind of features detection. Features detection refers to the techniques used to detect interesting portion in a picture like edges, corners or curves. A point of interest is a point in the image which is stable under image perturbations such as those that occur when the camera moves. There are multiple ways to detect such point: The most well known algorithms are SUSAN, Harris, Shi-Thomasi. I choose to use the Harris corner detector because this algorithm is easy to implement and there are a lot of examples of implementations. To implement this Harris detector, we acquire an image from the camera and then use OpenGL to draw this image as a texture that fills the window on which the scene is drawn. Then, in a shader for each pixel its neighbor pixels are read and a value is computed with the Harris detector and written to the frame-buffer. The result is then saved to a new texture. Once the Harris detector has created a new texture with corner in white, the program loop to each pixel of this new picture and add point where there is a white spot (white pixels represent high values and black pixel represent low values).

3.2 Motion blur

A big problem to handle is the motion blur. The motion blur is the integral of the picture during the time of exposure. As for all camera this time is not null, when an object moves, its color will be distributed along the direction of its motion as well as when the camera moves. This effect causes the disappearance of many features because this will decrease the value of certain derivatives which the features detector needs. A motion blur can be represented by a function that spread a points on the picture aka PSF (Point spread function). Such a function can be defined by a convolution.

![Figure 3: This picture shows the effect of a convolution with a point spread function on a picture with an object. (adapted from https://fr.wikipedia.org/wiki/Fichier:Convolution_Illustrated_eng.png)](image)

4 Harris detector

A Harris detector detects corners by making the assumption that at a border, there is a strong change in intensity in colors and then that if a strong change occurs in two directions there is a corner. Mathematically if we see the picture as a function of (x, y) a change in intensity in one direction represents the derivative in this direction. These derivatives are simply calculated by taking the differences between the pixels next to the current point. The computation of the derivative at the red pixel is then

\[
\begin{align*}
  d_x = 2 \cdot (I_c - I_d) + I_h - I_a + I_e - I_f \\
  d_y = 2 \cdot (I_b - I_q) + I_h - I_a + I_e - I_f
\end{align*}
\]

![Figure 4: This picture represents a schema of a pixel and its neighbors (a, b, c, d, e, f, g, h) to illustrate how the derivative is computed](image)
Where \( I \) represents the intensity of the pixel and \( d_x, d_y \) the derivative in \( x \) and \( y \).

To detect two large derivatives in different directions we construct the following matrix with the derivatives all around the current point:

\[
A = \sum_{x,y} w(x, y) \begin{bmatrix}
    d_x^2 & d_x d_y \\
    d_y d_x & d_y^2
\end{bmatrix}
\]

Where \( w(y, x) \) is a weighting function. If the eigenvalues are large positive, then the point is probably a corner. The Harris detector determines if this is the case by computing \( \det(A) - k \text{trace}(A)^2 \)

\[\text{Figure 5: This picture is the result image of the Harris detector. The intersection of a chessboard are clearly shown by a white spot.}\]

In the GLSL\(^{15}\)(GLSL is OpenGL Shading Language) part which runs on the GPU, the matrix \( A \) is constructed like this:

\[
\text{mat2 m = mat2(0); for (int i = 0; i < 8; i++)}
\]
\[
\text{m[0][0] += gx[i] * gx[i] * gw[i];}
\]
\[
\text{m[1][1] += gy[i] * gy[i] * gw[i];}
\]
\[
\text{m[0][1] += gx[i] * gy[i] * gw[i];}
\]
\[
\text{m[1][0] = m[0][1];}
\]

where \( \text{gx}[i], \text{gy}[i], \text{gw}[i] \) represents respectively the derivative in \( x \), the derivative in \( y \) and the weight which is computed with a Gaussian\(^{16}\) of the pixel position relative to the sub-image kernel\(^{17}\) and an experimentally fixed value.

5 Handling motion blur

Since the time of exposure of a camera isn’t instantaneous, the movement of the camera is integrated in the picture. This creates a blur. A trivial solution to avoid this problem is to decrease this time of exposure. But it’s not often possible and this time can’t be null. Another solution is to do a blind-deconvolution. A blind-deconvolution is the process to invert the effect of a convolution without prior knowledges of the convolution. (Here we want to invert the effect created by the motion blur). But there are faster methods. Sharpening the image to make corners more visible was tested but this process adds noise. If the sharpening is strong enough to avoid the blur on the corners, there are plenty of points that are added because of the noise and these points wouldn’t be stable as the noise change a lot between each frames. An other option is to determine the average direction of the motion blur and its intensity and then to compensate this move.

According to the paper by Xiaogang Chen\(^{18}\) the result of the motion blur is that a high-frequency decreases on the motion direction. On a picture, the high-frequency means pixels value that are rapidly changing in space. This is caused by the blending of colors over the blur direction. The authors of this paper indicate that the general motion blur direction is where the squared derivative is the smaller. Following the previously mentioned paper, the derivative of an image at the direction \( k \) degree from the horizontal plane is computed like this

\[
\begin{bmatrix}
    d_x \\
    d_y
\end{bmatrix}
\begin{bmatrix}
    \cos(k) \\
    \sin(k)
\end{bmatrix}
\]

Then on a kernel of size \( M \times N \) the function of the squared directional derivatives is \( J(k) = \sum_{x=1}^{N} \sum_{y=1}^{N} \left( \begin{bmatrix} d_x & d_y \end{bmatrix} \begin{bmatrix} \cos(k) \\ \sin(k) \end{bmatrix} \right)^2 \)

On the blur direction \( J(k) \) will be minimum. In my implementation, I use a kernel in a picture of size 32x32 pixels.

\[\text{Figure 6: This picture represents the direction of the motion blur at each sub-image kernel of 32 \times 32 pixel by a small red line.}\]

\(^{15}\)https://www.opengl.org/documentation/glsli
\(^{16}\)http://en.wikipedia.org/wiki/Gaussian_function
\(^{17}\)http://en.wikipedia.org/wiki/Kernel_(image_processing)

\(^{18}\)www.pami.sjtu.edu.cn/people/xgchen/chen_ICIP2010.pdf
6 How SLAM algorithms works

In all the SLAM-like algorithm the first part is the initialization. The first thing is to detect features that can be tracked from different points of view (Harris, Shi-Thomasi). Then these features must be detected from another point of view. Once we have multiple pictures with each their set of features (which are shared but appear at different position due to the different place of the camera) they are associated together such that we know that a given feature in a picture corresponds to the same world position on another image. For this we use different kinds of features descriptors\(^\text{19}\). In PTAM for each points of interest detected, a small patch of size 8x8 in the picture is stored and transformed with the estimated camera transformation before they are compared. Once there is a set of points that matches the points on the other image, it is then possible to find their position in the world.

Figure 7: This picture is a scheme of the matching points between two frames taken from different points of view where the transformation between each position is represented by the matrix \(T\)

If we know the transformation \(P\) that projects a point in the world to a point in the picture from the camera and the transformation \(T\) which describe the derivative of the displacement of the camera between two images, then we have a system of equations. The transformation that projects a point from the world to a point in the image acquired by the camera can be know by a calibration of the camera.

\[
P(x, y, z)^T = (x_1, y_1)^T
\]

\[
P \cdot T(x, y, z)^T = (x_2, y_2)^T
\]

From which we can know the position \(x, y, z\) of the point relative to the camera (stereo technique). Then a plane is computed and the position of the 3D points are computed relatively to the plane. Once the initialization is done, the position of the camera relative to the virtual plane is stored as the position of the point. In PTAM the structure that stores these information is called a key-frame. Then the displacement of the camera is estimated using an algorithm to find the camera transformation between two camera poses (usually 5-points algorithm\(^\text{20}\)) or 8-points algorithm\(^\text{20}\). The error is then corrected using the previous generated key-frames. New key-frames are added once the precision is above a defined threshold using an heuristic (number of points used for tracking for example). Adding new key-frame is done in the mapping part while finding interest point and finding their position is done in the tracking part. Generally both works together in one process but in PTAM they run in parallel in two different threads. The points of interest on each key-frames are then reprojected and all key-frames are reajjusted.

Figure 8: This is a schema that represents the keyframes \(K\) and their associated set of points

The position of the camera is not perfect and need to be adjusted prior to previous key-frames. In PTAM this process is called Bundle adjustment. This is where PTAM use the second order minimization. When the camera don’t view to any of the known environment this error will always be high and this is exactly what we use to determine if the scene must be drawn or not.

Figure 9: This schema represents the correction between the estimated transformation and the real transformation.

This process continues to track point and to update the camera position and to minimize the error as the program runs.

\(^{19}\text{www.cs.toronto.edu/~kyros/courses/2503/Handouts/features.pdf}\)

\(^{20}\text{https://en.wikipedia.org/wiki/Eight-point_algorithm}\)
PTAM is an augmented reality software developed at Oxford university. It come with an example of a 3D augmented scene.

### 7.1 Pros/Cons

To have an idea about how it matches our needs, I tested it in a regular room with a low-end Logitech camera C170 on a regular laptop. I tried different environments, watching on a table with nothing on it, on a table with many objects, on the ground and for each environment I move the camera to many different orientations. I tried to hide the objective of the camera and to orient the camera to another place and notice how PTAM handles these cases. Another test is to view the augmented scene from a very close distance to the scene and then from a very far distance. What we observed is that PTAM can have some difficulties to derive a precise estimation of the distances. When the scene is watched for a distance of less than 10-15 cm, the scene will not stay where it should be and will jump at different places in the field of view. In the other case until a distance of 3m the scene is drawn without problem as long as the detected features are still apparents in the picture taken from the camera. A good point is that when PTAM lost its "view" it is capable to recover the landmarks instantly when the camera is watching back to the scenery.

PTAM needs to detect features in the room, then the scene will move between multiples position in a completely empty room or if there is not so much contrast.

An other problem is that PTAM was created to run in small area. This means that the points of interests lie only in a small area and once it have a sufficient number of these, it would not add any more new points. A solution to this problem is to use multiples maps or to increase the number of points that can added. There is an implementation called PTAMM which is a modification of PTAM using multiple maps.

As PTAM uses only a single camera it can’t have a precise information about the size of the objects. The factor that can have an effect on this is the displacement of the camera during the creation of the map by stereo at the initialization phase.

---

Figure 10: This is a screenshot of PTAM during the initialization phase.

The last problem is the scene which is drawn even when the camera doesn’t point to the known map. For example the user is watching toward the augmented scene and then he watches behind himself and see the scene he was seeing before appears from nowhere.

Figure 11: This picture is taken when the camera is watching toward the scene.

Figure 12: This picture is taken when the camera isn’t watching to the scene anymore.

---

21http://www.robots.ox.ac.uk/~gk/PTAM/
22http://www.ox.ac.uk/
23http://www.logitech.com/de-ch/home
24http://www.robots.ox.ac.uk/~bob/research/research_ptamm.html
Multiples solutions were considered to tackle this problem. It is possible to detect the angle of view with the matrix of transformation provided by PTAM and then, when we know we are out of the scene we search for a pattern and until this pattern is found we do not draw the scenery. Or we can use external sensors to acquire external information about where we are watching. Note that it is not possible to simply relies on angles to determine if the scene is visible because the transformation are determined with the key-frames and a matrix is created from an iterative algorithm. The solution taken is to draw only if the score returned by the ESM algorithm is high enough. ESM stands for Efficient Second order Minimization which is an algorithm for homography transformation. This algorithm minimize the sum-of-square differences (SSD) between a reference template and the current template. In PTAM it’s used to estimate the transformation parameters. To determine at which threshold we can draw the scene, I just printed the score obtained by the ESM and tested experimentally what values were obtained when facing toward the scene and toward other direction. This give good results, since the score varies with a factor 10 between both cases.

8 PTAM design

PTAM is programmed in C++ and relies on a library called libCVD also from Oxford. libCVD\(^{25}\) is a library for computer vision but unlike OpenCV\(^{26}\) it is designed to be small and fast instead of being a complete framework. PTAM has two threads, one is doing the tracking part while the other is updating the map.

While the tracking system continues to detect features, the mapping add new keyframes. Integrate new points when they can be associated between the keyframes and performs adjustment over the existing keyframes.

\(^{26}\)http://opencv.org/

Figure 13: This is the shema that we can find in the paper of the developers of PTAM (http://www.robots.ox.ac.uk/~gk/publications/KleinMurray2007ISMAR.pdf). It shows clearly how the mapping work.

9 Adaptation of PTAM for a custom scene

PTAM already provide a simple scene EyeGame but it’s possible to change this scene for another. The provided scene has its own class. This scene is instantiated in ARDriver and all of its functions are called here. It is then easy to rewrite a new class that use OpenGL to draw our own AR scene. I created a small system that can be easily used to create a Roombots interface. I added a mesh\(^{27}\) loader to load a Roombot mesh component. The mesh loader load mesh in the Obj\(^{28}\) format.

\(^{27}\)a mesh describe a 3D object with 3D vectors and textures coordinates to be applied to it
\(^{28}\)http://www.martinreddy.net/gfx/3d/OBJ.spec
Appendices

Details on the PTAM source code

ARDriver
- Draws a plane.
- Creates an instance of the AR scene, calls its drawing function with the camera inverse transformation as argument.

ATANCamera
- Calibrates the camera and computes the projection matrix.

Bundle
- Manages Keyframes.
- Minimizes the error in the transformation matrix between key-frames.

CalibCornerPatch, CalibImage
- Detects chess corner for the calibration phase.

CameraCalibrator
- It is the tool which permits the user to calibrate the camera.

EyeGame
- The provided initial AR scene.

GLWindow2
- HUD, input events.

GLWindowMenu
- Display application menus.

HomographyInit
- Compute an approximation of the camera transformation between two points of view.

Keyframe
- Creation of new Keyframes.
- Store set of points associated with a camera position.

Map
- Store MapPoint and delete bad points.
- Store Keyframes.

Map Maker
- Create the initial map with stereo initialization.
- Recompute keyframes’s transformation matrix to correct the errors.
- Generate new keyframes.
- Look for shared points between keyframes to add them in the current map.

MapPoint
- Store a world position of a point.
- Store a pointer to the position of the image patch where the point appear.

Map Viewer
- Simply draw a map in view map mode.

MEstimator
- Implementation of a MEstimator 29.

MiniPatch
- It is a small portion of the image.

PatchFinder
- It is a features-descriptor used to compare a new points with previous one.

Relocaliser
- Estimate a camera rotation.

Shi Thomasi
- Detect corner with a modified harris detector.

SmallBLurryImage
- This create a small copy of the image for a keyframe. Use by the relocalizer.

SmallMatrixOpts
- Tools for basic operations on matrix such as matrix inverse.

System
- Grab the videos, launch threads and main loop.

Tracker
- Manage all the tracking procedure.

VideoSource

• Open the video device and provide video source.

Image acquisition
In my own implementation the image from the camera is taken with Video4Linux in the pixel format yuv422. This format store 4 byte per 2 pixels. Gray-scale information for each pixel and color information for the pair. The image is then converted to RGB for display and to 1 byte gray-scale for tracking. Note that PTAM and many others computer vision systems do the same.