
MARKER DETECTION USING AR DRONE AND OPENCV

Muhammad Hassan Nadeem, Danial Nawaz

Department of Electrical Engineering

SBA School of Science & Engineering, LUMS, Pakistan

{15100063, 15100177}@lums.edu.pk

ABSTRACT

The aim of this project was to explore embedded applications of the AR Drone 2.0. Traditionally the Drone has been used with ROS running on a Linux machine. Being tethered to a PC with a wifi link seriously limits its applications. This project discovers the possibilities of using AR Drone as an aircraft that can navigate autonomously, without human control and beyond line of sight. As a proof of concept the Drone was made to follow a marker.

The project starts with cross compilation of OpenCV, FFmpeg and x264 libraries for the Drone. This is so that our program can access and process the camera feeds of the Drone for autonomous flight. Next C++ Boost libraries are cross-compiled for the Drone to make a multithreaded application that can be used for robust control of the Drone. Finally an application is written in C++ that uses OpenCV to detect markers in the camera feed and makes use of Drone's API to implement a PID control aimed to make it track a marker.

RELATED WORK

This project is first of its kind. There has been no documented attempt to make an autonomous embedded application for AR Drone. Most of the projects done on AR Drone makes use of AR-Drone Autonomy package [1] for ROS. This package provides a wrapper around AR Drone's API, for the control of the Drone using a ROS powered machine.

This project takes hints from ROS's AR-Drone Autonomy package [1] and AR Drone SDK [2], and uses AR Drone API to develop an embedded application for the Drone.

OUR APPROACH

Cross Compilation of OpenCV + FFmpeg + x264

Major part of this project involved cross-compiling OpenCV for ARM processor of AR Drone. The Drone's firmware sends the video feeds of the two cameras on TCP Port 5555 encoded in x264. This encoded stream is decoded with FFmpeg and x264 libraries. OpenCV extracts frames from this stream and using algorithm explained in next section extracts the marker.

Everything related to the cross compilation of OpenCV and accompanying libraries is extensively documented and published on my blog for the benefit of others. The links below provide in depth information starting from setting up cross compilation environment and ending with running an OpenCV project on AR Drone.

1. [Cross-Compilation for AR Drone](#)
2. [Cross-Compile OpenCV with FFmpeg \(x264 and xVid\) for AR Drone 2.0 \(ARM Processor\)](#)
3. [Cross Compiling OpenCV project for ARM, AR Drone 2.0](#)
4. [Cross Compilation for AR Drone Made Easy](#)
5. [Running Cross Compiled OpenCV project on AR Drone](#)

Marker Detection

Next step in achieving our goal was to develop a robust code to track the desired marker. We started off with ready-made libraries for the detection of QR code but its detection was computationally very expensive for AR Drone's small arm processor so we decided to make our own very simple marker. In order to detect the code we followed the following steps.



Figure 1: Our Simple Marker

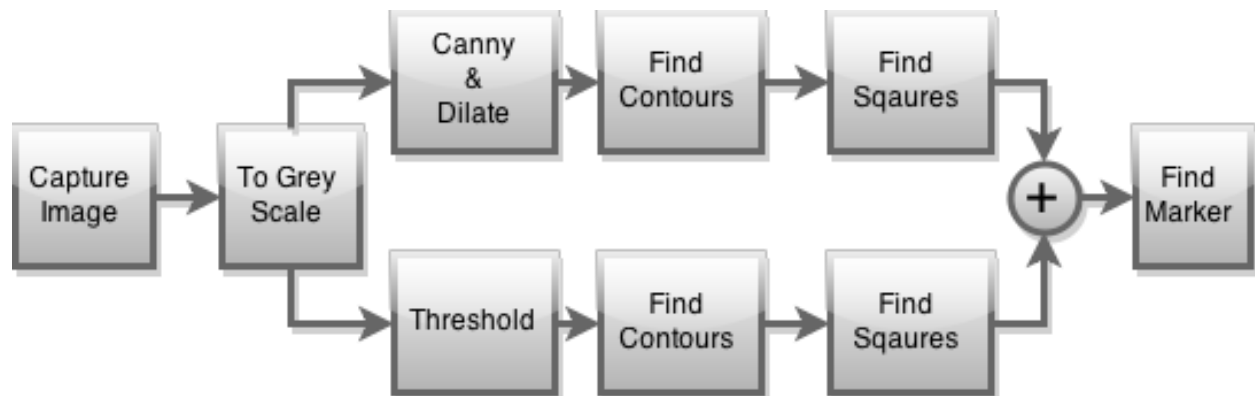


Figure 2: Block Diagram showing the process of marker detection

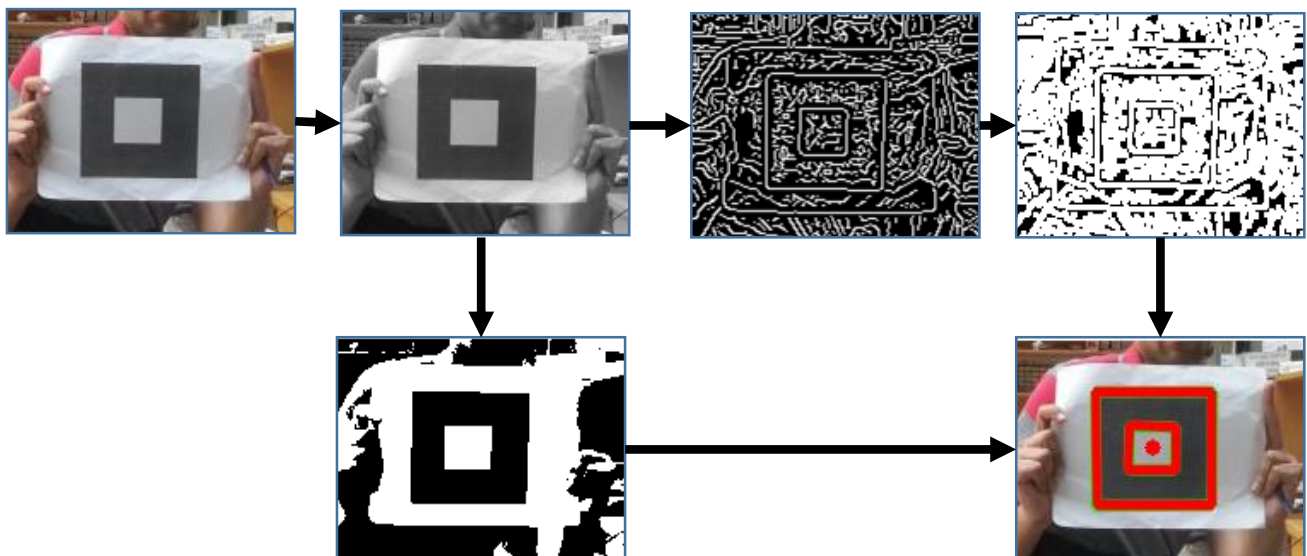


Figure 3: Marker Detection Algorithm in Action

Navigation Data

The navigation data is a mean given to a client application to receive periodic (With less than 5ms period) information on the drone status. Navigation data includes information about Drone's status like roll, pitch, yaw angles, altitude, x, y z velocities etc.

The navigation data are sent by the drone from and to the UDP port 5554. Information are stored in a Binary format and consist in several sections blocks of data called options. Each option consists in a header (2 bytes) identifying the kind of information contained in it, a 16-bit integer storing the size of the block, and several information stored as 32-bit integers, 32-bit single precision floating-point numbers, or arrays. All those data are stored with little-endianness.

Header 0x55667788	Drone state	Sequence number	Vision flag	Option 1			...	Checksum block		
32-bit int.	32-bit int.	32-bit int.	32-bit int.	id	size	data	...	cks id	size	cks data
32-bit int.	32-bit int.	32-bit int.	32-bit int.	16-bit int.	16-bit int.	16-bit int.	16-bit int.	32-bit int.

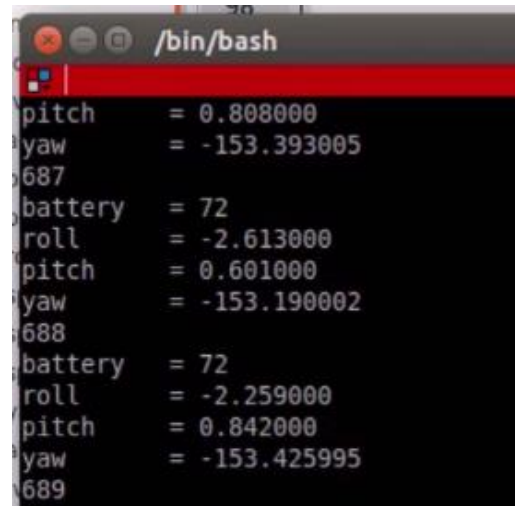
Initiating the reception of Navigation Data

Wireshark was used to sniff packets to and from AR Drone to find out the protocol to initiate the reception of Nav Data. To start receiving Navigation Data the client must send a series of bytes on port 5556. After initialization the drone is in bootstrap mode, in this mode the drone only sends the status of the drone. To get the drone to publish complete navigation information an AT command:

"AT*CONFIG=\"general:navdata_demo\\\", \"TRUE\\\"\\r"

followed by another AT command: "AT*CTRL=0" must be sent. After this the Drone will start sending encrypted navigation data stream every 5 ms. There is a

need to send clear watchdog timer AT command to the Drone every 50 ms, otherwise the Drone will consider the connection with the client to be lost and will stop sending data after 2000 ms.



```

/bin/bash
pitch = 0.808000
yaw = -153.393005
687
battery = 72
roll = -2.613000
pitch = 0.601000
yaw = -153.190002
688
battery = 72
roll = -2.259000
pitch = 0.842000
yaw = -153.425995
689

```

Receiving and Decoding Navigation Data

All the processing related to receiving and decoding the navigation data is done in a separate dedicated thread. The process starts from sending special byte sequence of {0x01, 0x00, 0x00, 0x00} to the navigation data UDP port 5556, and getting the Drone out of bootstrap mode so that it published complete navigation data. Rest of the work is done in an infinite while loop that receives the byte stream of the navigation data, decodes and extracts the navigation data and publishes the result in a special C++ struct containing all the useful information.

Multithreading

Since there is a need to clear watchdog timer every 50 ms and receive navigation data every 5 ms. There was a need for multi-threaded application that would take care of all this and process camera feed and issue velocity commands simultaneously. For that Boost libraries were cross compiled for the Drone.

Boost.Thread enables the use of multiple threads of execution with shared data in portable C++ code. It provides classes and functions for managing the threads themselves, along with others for synchronizing data between the threads or providing separate copies of data specific to individual threads.

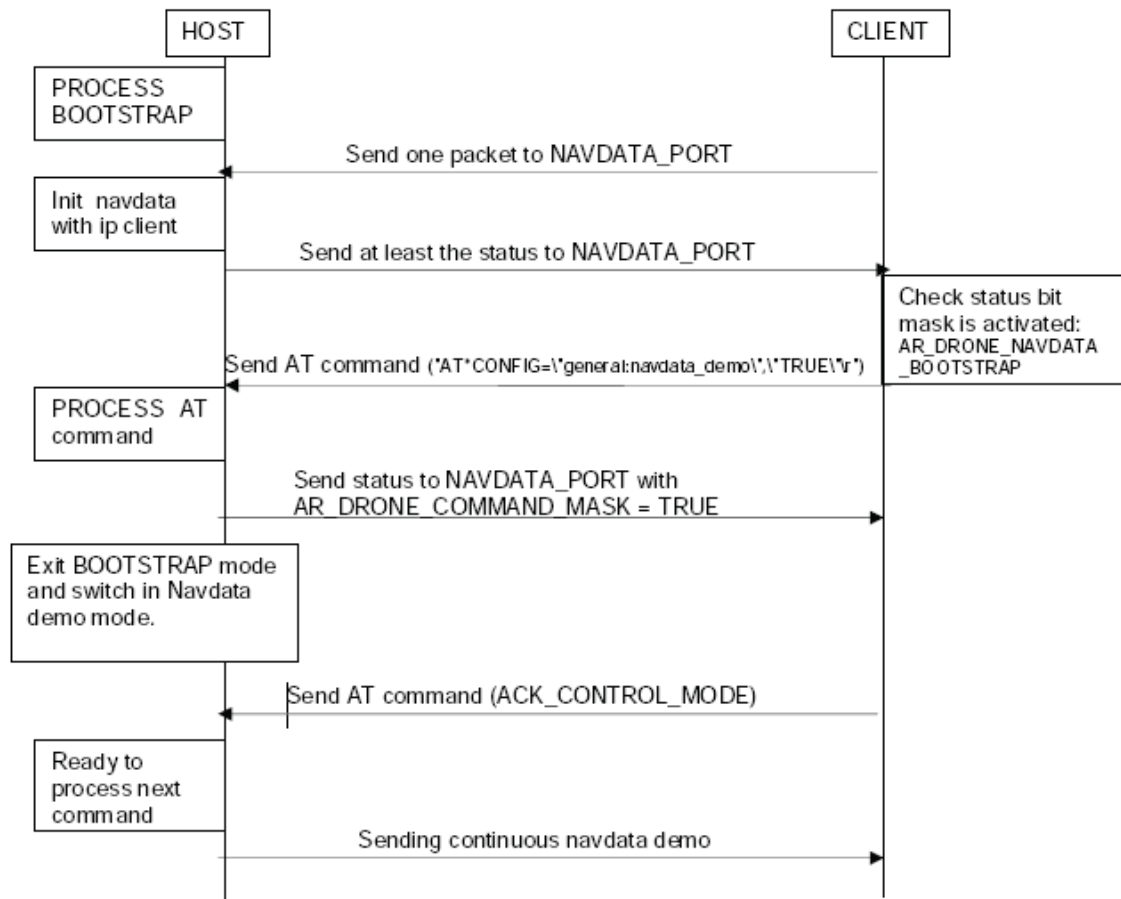


Figure 4: Navigation Data Stream Initiation

Control

The Drone is made to follow the marker using 3 separate PID controller. After the marker is detected, center point of the marker is found. An error which is the difference between the position of the center point of the marker and center of the image frame is calculated. This error is decomposed into up-down and left-right errors which is fed into 2 PID controllers that control the up-down and left-right errors. The third error i.e. the forward and backward error is calculated using the square root of the area of the outer square of the marker. The third PID controller works to minimize this error. The resultant controller from this approach is a very robust controller that can minimize error in 3 degrees of freedom, this results in tight tracking of the marker by AR Drone.

Control using AR Drone API

The Drone firmware is a 'program.elf' what runs every time the Drone is booted. It listens on UDP port 5556 and accepts AT commands to control the action of the Drone. This program provides an application program interface and a higher layer of abstraction to control the drone. Traditionally a laptop is connected with the drone via wifi and all the communication is done with host with ip

address 192.168.1.1. ROS driver for AR Drone and mobile phone applications for the Drone all control the drone in a similar fashion. We have written a C++ application that connects to the localhost from within the Drone and makes use of this already implemented API. An alternative of this approach would be to kill the program.elf and control the individual motor drives. This approach would both be an unnecessary work and may not compare with the quality of the Drone's firmware. Therefore instead of reinventing the wheel, we have used the firmware provided API to control the drone from within the drone.

AT command	Arguments ¹	Description
AT*REF	input	Takeoff/Landing/Emergency stop command
AT*PCMD	flag, roll, pitch, gaz, yaw	Move the drone
AT*PCMD_MAG	flag, roll, pitch, gaz, yaw, psi, psi accuracy	Move the drone (with Absolute Control support)
AT*FTRIM	-	Sets the reference for the horizontal plane (must be on ground)
AT*CONFIG	key, value	Configuration of the AR.Drone 2.0
AT*CONFIG_IDS	session, user, application ids	Identifiers for AT*CONFIG commands
AT*COMWDG	-	Reset the communication watchdog
AT*CALIB	device number	Ask the drone to calibrate the magnetometer (must be flying)

Figure 5: AT Commands List

Video Stream

AR.Drone 2.0 use H264/MPEG baseline profile for high quality video streaming. The following parameters can be adjusted for the live stream:

- FPS : Between 15 and 30
- Bitrate : Between 250kbps and 4Mbps
- Resolution : 360p (640x360) or 720p (1280*720)

For our project we chose a stream with 15 FPS, 500 kbps, 360p, encoded in H264, of which every 10th frame was processed owing to the time taken to extract the marker from each frame.

The video stream is transmitted on TCP socket 5555. The Drone will start sending frame immediately when a client connects to the socket.

To start receiving the video stream, a client just needs to send a UDP packet on the drone video port. Again there is a need to need reset the watchdog timer every 50 ms otherwise the Drone will consider the connection lost and stop the stream.

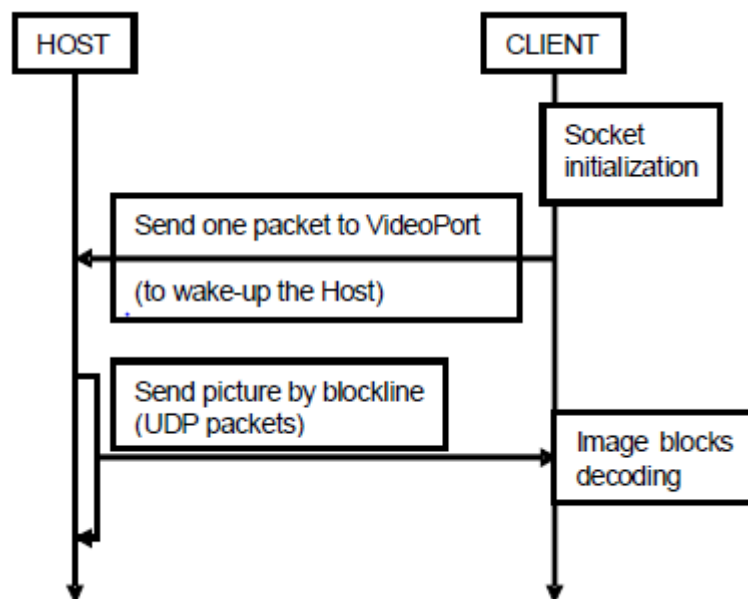


Figure 6: Initiating the video stream

EXPERIMENTS & ANALYSIS

Our initial experiments with the AR Drone resulted in very slow response times with delays of more than 1s between each frame. The code was optimized and the processing time of each frame was brought down to 0.7s which was still very large for our application.

Function	Time Taken (seconds)
Image acquisition (background)	0.01
Greyscale	0.01
Canny + Dilate	0.09
Find squares in Canny	0.31
Threshold x 6	$0.01 \times 6 = 0.06$
Find squares in Threshold	$0.03 \times 6 = 0.18$
Find Marker	0.02
AR Drone Control + PID	0.02
Total: One iteration of while loop	0.70

After careful profiling it was discovered that finding the marker using canny is the most computationally expensive and time consuming part of the algorithm. And removing canny would decrease the time to process one frame by as much as $1/3^{\text{rd}}$ to approximately 0.25 ms per frame. But this comes at a cost. Without canny the algorithm works reasonably well but fails to detect the marker in uneven lightning conditions.

FUTURE WORK & CONCLUSION

All in all this project was a huge success. We were successfully able to provide a proof of concept embedded application for AR Drone. Although the project we chose to demonstrate this did not turn out be best of the choices, given the limited processing capabilities of the Drone. The marker detection algorithm can certainly be fine-tuned and optimized for low speed processors. But the Drone will find its true application in projects where image processing is less frequently required. These projects may include data collection along the roads/highways and cannals.

Our work on this project has opened new doors for further research work involving AR Drone.

REFERENCES

- [1] Autonomy Lab - ardrone_autonomy. Available: https://github.com/AutonomyLab/ardrone_autonomy
- [2] AR Drone 2.0 – SDK 2.0.1 [Online]. Available: <https://projects.ardrone.org/projects/show/ardrone-api>