Autonomous Navigation to Multiple Landmarks with a Quadrotor Aircraft

Cristan Gonzalez*, Holger Rasmussen†, Barbara Sepic‡ and Felix Stahlberg§
*T00902102, Email: crisgonzalez346@gmail.com
†A00922103, Email: h.rasmussen@gmx.net
‡A00922104, Email: barbara.sepic@gmail.com
§TA00922105, Email: fstahlberg@gmail.com

***Abstract*—In this work, we address autonomous path planning and tracking in a 3D space using an aircraft. We use shape detection, object depth calculation and dynamic navigation planning to navigate a quadrotor through a set of rings. We apply our results in an AR.Drone application. This drone became popular in recent years for educational purposes in robotics. Besides challenges mentioned above, we had to face unstable control of slow and percise movements and inaccurate sensing.**

I. Introduction

II. AR.Drone Quadrator

The AR.Drone is a quadrotor, an aircraft with four propellers. It is a UAV that is primarily used as toy for children to play with, but has become popular with researchers and students of robotics due to availability and low price of the AR.Drone in comparison to other UAV in the market. It is powered from a rechargeable Lithium polymer battery pack delivering 11.1 V, 1000 mAh. With a weight of 380 g or 420 g (with the ”indoor hull”) it can maintain flight for about 15 minutes when the battery is fully charged. [1].

The AR.Drone has two cameras, one on the bottom of the AR.Drone, know as the vertical or bottom camera, and the other on the nose of the AR.Drone, know as the horizontal or frontal camera (see Section 1). The horizontal camera has approximately $75° \times 60°$ field of view and provides 640×480 pixel color image [1]. The AR.Drone also contains two sonar sensors located to the left and right of the bottom camera to measure its altitude. Communication between the AR.Drone and the client application is possible through a built in Wi-Fi. Different ports are used for sending video and navigation data from the AR.Drone and receiving the commands from the client.

Since the AR.Drone is a quadrotor, there is usually a need to be careful with the delicate system of how to control the quadrotor. Due to the control board of the AR.Drone, there is not much to worry about the physics of a quadrotor. In order to take full advantage of flight, the speed of the propellers must be the same and certain propellers must rotate either clockwise or counter clockwise [3]. Other things that the control board of the AR.Drone does as well is assist with the landing and take off of the AR.Drone [1]. All that the program has to do is send commands to the AR.Drone to land or takeoff instead of using machine code to program each propeller and have it be very precise in order to avoid crashes that would severely damage the AR.Drone. The control board of the AR.Drone also has some safety features that help prevent the blades of the propeller from getting damage. If any foreign object touches one of the blades, it makes the blade either not move or move backwards, the control board will stop the other motors of the propeller and forcefully land the AR.Drone. The vertical camera image is processed to estimate the drone speed relative to the ground [1].

A. Navigation Data

The AR.Drone API has some software implemented to help for data acquisition and control of the AR.Drone. The Navigation Data, or NavData as it is called in the AR.Drone, is a mean given to a client application to receive periodically (¡ 5ms) certain information on the drone status [3]. We use following information fetched from these packages.

- Roll, Pitch, and Yaw measurements (see Section 2)
- Altitude
- Horizontal speed values in x- and y- direction (fusion of the output of an optical flow algorithm and the orientation of the drone)
- Control state and battery level

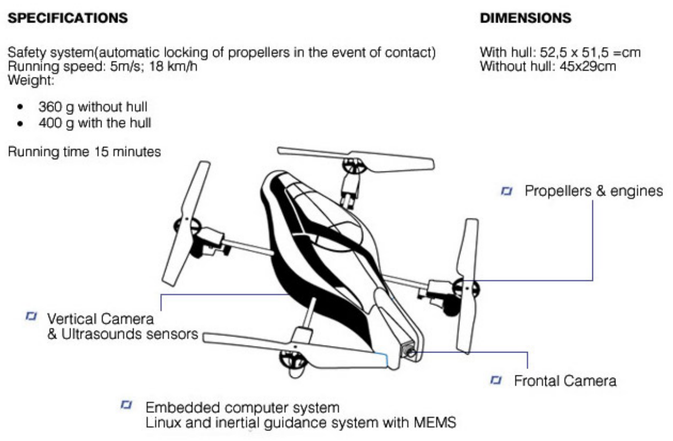
SPECIFICATIONS
Safety system(automatic locking of propellers in the event of contact)
Running speed: 5m/s; 18 km/h
Weight:
360 g without hull
400 g with the hull
Running time 15 minutes
DIMENSIONS
With hull: 52,5 x 51,5 =cm
Without hull: 45x29cm
Propellers & engines
Vertical Camera & Ultrasounds sensors
Frontal Camera
Embedded computer system Linux and inertial guidance system with MEMS

Fig. 1. The AR.Drone.

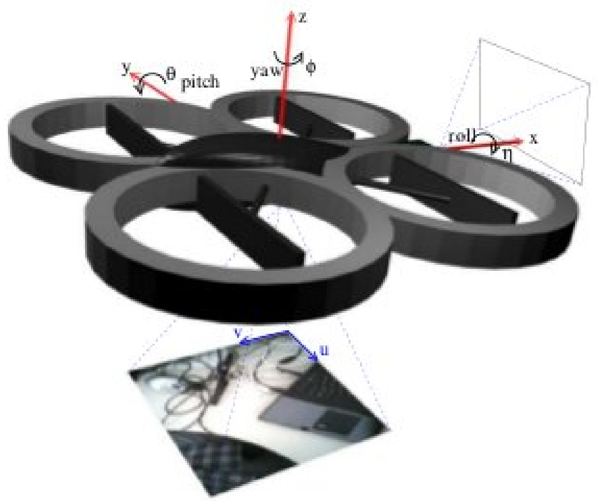

Fig. 2. Roll, pitch, and jaw on the AR.Drone.

B. Video Camera Capturing

The stream channel provides images from the horizontal and/or vertical cameras. The horizontal camera image is not provided in the actual camera resolution, but it is scaled down and compressed to reduce its size and speed up its transfer over Wi-Fi. As a result, the external computer obtains a 320 240 pixel bitmap with 16bit color depth source 1. The disadvantage of the camera system is that a user cannot obtain both camera images at a time. Instead, the user has to choose either the horizontal camera or the vertical camera. It is also possible to switch between cameras, but the process of switching cameras is not done instantaneously; it will take approximately 300 ms to change camera modes and during the process of the camera switch, the images that is given to the user are invalid data source 1. However, in our work only the front camera is used.

III. Project Architecture

Figure 3 depicts the basic structure of the project architecture. Our program is based on the hybrid architecture design paradigm. It can be divided into two parts: deliberative and reactive layer. In our case, a final state machine has a central position in the deliberative layer. It describes the current status of the drone and defines the set of behaviors in the reachtive layer. The reactive layer consists of reactive behaviours which respond to the currently captured video or received navigation data (see Section II-A). Additionally, the planner controls the input word of the final state machine. Note that this input word is generated online, since some information is not available at the begining. For example, the planner does not know how many rings exist in the scenery until the end of the `BuildWorldModel` state. It extends the input word of the final state machine at the end of this state.

A. Planner

Planner observes and plans the whole program. Its logic is implemented in `Planner`. Depending on the results it

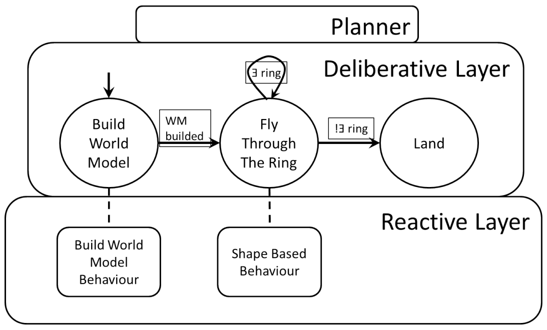

Fig. 3. The basic structure of project architecture.

switches between the stages and defines their inputs. To be more percise, it actually defines the inputs of the behaviours which will be active in this state (as defined in `StateManager` - see Section III-B). For instance, depending on the number of the rings it got from the first state (`BuildWorldModel` State) it decides how often it should repeat the `FlyThroughTheRing` State. Furthermore, it also defines the order in which the rings are passed (the nearest to the farest).

B. Final State Machine

The implementation of the final state machine and the behavior dependencies on its states can be found in `StateManager`. It defines which behaviours are active in which state. Furthermore, it notifies behaviors when a new video frame is available or a new NavData package was sent. This enables the behaviors on the reactive layers to react to perceptions immediately. As shown in the Figure 4 our final state machine consists of three states. After building the world model, `FlyThroughTheRing` State is repeated until it flies through all the rings recognized in the first state. Finally, the AR.Drone should land. The details of each state are described below.

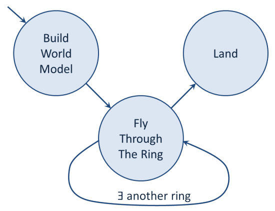

Fig. 4. Final State Mashine.

C. Behaviours

As we already mentioned, behaviours react to received data (Navdata packages and/or video images). Depending on this inputs, they adjust their actions. Furthermore, the whole logic of each state is implemented in its state(s). Functionalities of implemented behaviours are described bellow.

IV. Implementation

A. `BuildWorldModel` State

The `BuildWorldModel` State recognizes and defines the rings with their positions. Its logic is implemented in `BuildWorldModelPilot` and described in Section IV-B.

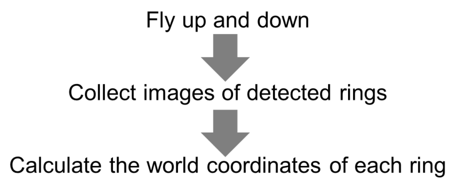
Fly up and down
Collect images of detected rings
Calculate the world coordinates of each ring

Fig. 5. Flow chart for the `BuildWorldModel` State.

B. `BuildWorldModelPilot` Behaviour

As the name indicates, this behaviour is active in the `BuildWorldModel` State (see Section IV-A). It depends on the images received from the frontal camera. Our goal is to capture enough images of the rings, so we can calculate their depths (and positions) percisely. Therefore, we fly up and down capturing the images of detected rings along the way as shown in Figure 5. The description of the depth calculation can be found in Section VI. Just before ending this behaviour we sort all the objects with their calculated positions depending on their depth.

C. `FlyThroughTheRing` State

In the `FlyThroughTheRing` state the AR.Drone flies towards and through the ring. Its logic is implemented in `ShapeBasedPilot` and described in Section IV-D.

D. Shape Based Pilot

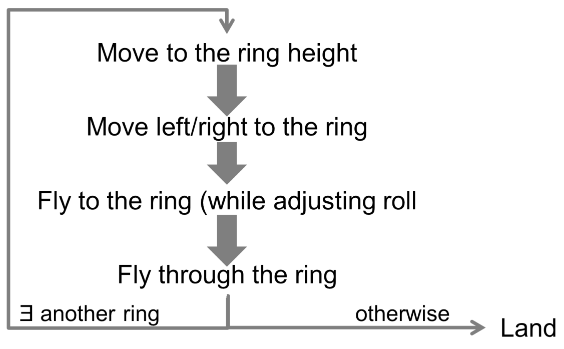

Fig. 6. Flow chart for the `BuildWorldModel` State.

The shape based pilot is a behavior used in the `FlyThorughTheRing` state (see Section IV-C). Figure 6 illustrates the control flow during this state. This behavior gradually adjusts the position of the drone to approach the ring in a beneficial angle and then fly through it. To initialize the pilot, it receives an action object from the planner, which defines the parameter of its mission. This action includes the target shape (e.g. red color, ring). The pilot will go through three phases to execute the action:

- Approach the ring
- Flying into the ring
- Flying through the ring.

Throughout the flight, shape recognition is used to detect rings in the camera image (see Section V). The result is a `Shape` object, which stores a rectangle with x, y, $width$ and $height$ of the target ring as pixel positions. Calculating the offset between the image center and the center of the image gives hints about how to correct the current path of the drone in order to fly through the ring.

Approach the ring phase: In the first phase, the pilot needs to move towards the point, where the target ring is in the center of the frontal camera image. If the ring can not be seen from the current point of view, the drone adjusts its position depending on the previously calculated world coordinates of the target. The flying height can be accurately by turning the gaz as long as the drone and target z-coordinates differ. After adjusting the drone altitude, the drone flies to the right or left (depending on the y-coordinate of the target) until the shape centroid is horizontally centered in the camera picture. Then the drone flies in a constant forward movement towards the ring and compares the image center with the recognized shape in each frame. Differences in x-values are compensated by horizontally movements (roll). Vertical movements (gaz) compensate for differences along the y-axis in the image plane. Once the x and y values of the shape and the image center are aligned, the pilot moves on to the next phase.

Flying into the ring phase: During the flying into the ring phase, the pilot flies the AR.Drone forward until the ring cannot be seen and thus detected anymore. During this time, phi and gaz values are adjusted, so that the ring is always in the center of the image. Similar as in the initializing phase we compare the detected shape with the center of the image. However, in this phase the AR.Drone is still moving forwards, thus depending on the remaining distance and the amount of offset to the position at the end of the previous phase, the adjustments need to be more or less strong. In general, the greater the offset the stronger the adjustment and vice versa.

Flying through the ring phase: In the flying through the ring phase, the pilot should only move straight forward through the ring and stop once it is fully through. However, during this period the ring cannot be detected anymore, thus knowing when the AR.Drone has passed the ring is not easy or perceptable. The pilot assumes that after flying straight for some time, the drone has passed the ring. In practice, this takes about 2 seconds with $theta = -0.05$ (pitch). Furthermore, only moving forward is not ideal in practice because the AR.Drone may unintentionally change its yaw value and the phi value had to be adjusted to correct this issue. Therefore, we use the last known phi value and reduce it by half for every video frame during the phase. Once this phase is completed, the pilot will notifiy the planner, that the action has been completed.

E. `Landing` State

In the `Landing` state, the AR.Drone safely lands. As the AR.Drone has a build-in landing function, this state was trivially implemented by passing-through the control flow to the AR.Drone SDK.

V. Shape recognition

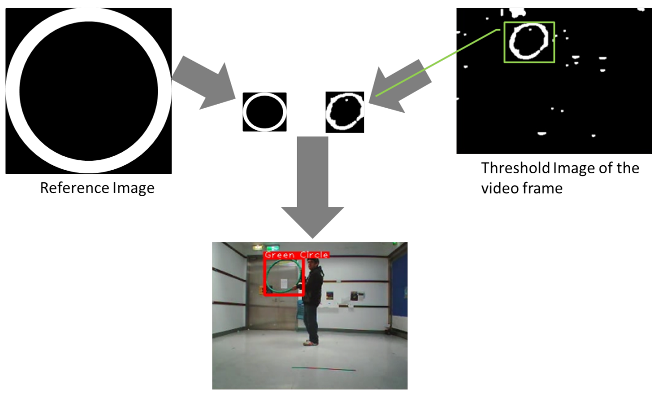

Fig. 7. Shape detection approach.

Shape recognition is an essential part in our program. It allows us to separate regions of obstacles from other elements in the environment. We use the OpenCV library 2.1 [5] for image processing. Our approach is shown in Figure 7. We are able to distinguish between two colors: red and green. For each color, we process the following steps for each frame in order to recognize shapes (here, illustrated for the color red):

1) Create a binary threshold image for the image. For each pixel, we set it to white if the blue and the green channel has a value not greater than the red channel minus a certain constant. Otherwise, we set this pixel to black.
2) We process the image with a closing operation. The pattern we use is a 12x12 circle pattern suitable in particular for ring shapes like in our case.
3) We add a dilation operation with a small pattern. This has been empirically proven to enhance the recognition performance because then the shape is more likely connected, which is required by the next step.
4) We use the OpenCV function `cvFindContours()` to find white regions in the image.
5) We store a reference image for each shape. For each region, we scale the reference image to the dimensions of the region and count misplaced pixels. Note that scaling the reference image generally comes along with changing the height-width ratio. Thus we are able to detect rings even if they are not parallel to the image plane.
6) If too many pixels are misplaced, we discard this region. Otherwise, we propagate a shape recognition in this region.

This approach allows us to rely on a quite stable and accurate shape recognition, despite noisy camera images. Turning the threshold of maximum number of misplaced pixels allows us sometimes to detect rings even if they are only partially visible or their region is disconnected in the threshold image.

VI. Calculating the depths

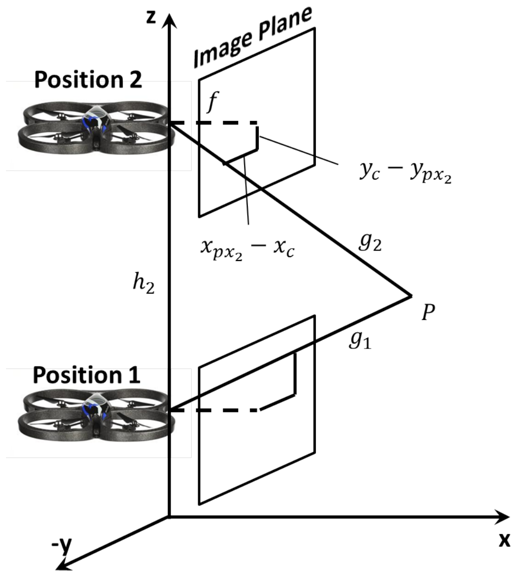

Fig. 8. Calculating the world coordinates of obstacles.

Our aim in the `BuildWorldModel` state is to calculate absolute positions and orientations of the obstacles in world coordinates. Figure 8 illustrates our approach. Using the front camera of the drone from n different heights h_i ($i = 1..n$) above the origin in the same orientation gives us a set of images. Let us assume we identify the pixel positions $(x_{px}, y_{px})_i$ of the projections of a distinct matching point $P = (x_P, y_P, z_P)$ in the world coordinate system for each image. We approximate x_P, y_P, and z_P using $(x_{px}, y_{px})_{Pi}$, h_i and the intrinsic parameters of the camera.

It turns out that for each image, we can form an equation describing a straight line through P by using basic vector arithmetics (assuming the pinhole camera model):

$$g_i := \begin{pmatrix} x_{si} \\ y_{si} \\ z_{si} \end{pmatrix} + \lambda_i \cdot \begin{pmatrix} x_{di} \\ y_{di} \\ z_{di} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ h_i \end{pmatrix} + \lambda_i \cdot \begin{pmatrix} f \\ x_{px_i} - x_c \\ y_c - y_{px_i} \end{pmatrix} \quad (1)$$

where f denotes the focal length of the front camera and (x_c, y_c) is the pixel position of the image center. We transform these equations into a overestimated linear system $Ax = b$ ($A \in \mathbb{R}^{3n \times (3+n)}$, $x \in \mathbb{R}^{3+n}$, $b \in \mathbb{R}^{3n}$) to calculate their point of intersection (which is equal to P). We show this transform for the case $n = 2$, but it is easily extendable to more than two equations.

$$A = \begin{pmatrix} 1 & 0 & 0 & -x_{d1} & 0 \\ 0 & 1 & 0 & -y_{d1} & 0 \\ 0 & 0 & 1 & -z_{d1} & 0 \\ 1 & 0 & 0 & 0 & -x_{d2} \\ 0 & 1 & 0 & 0 & -y_{d2} \\ 0 & 0 & 1 & 0 & -z_{d2} \end{pmatrix}$$

$$x = \begin{pmatrix} x_P \\ y_P \\ z_P \\ \lambda_1 \\ \lambda_2 \end{pmatrix}$$

$$b = \begin{pmatrix} x_{s1} \\ y_{s1} \\ z_{s1} \\ x_{s2} \\ y_{s2} \\ z_{s2} \end{pmatrix}$$

Solving the system $Ax = b$ with the least-squares method using the linear algebra package Armadillo [4] gives us the approximated coordinates of P.

In practice, we calculate the world coordinates of four points for each obstacle: the top most, the right most, the bottom most, and the left most. Calculating the intersection of the connecting vectors with similar methods as above gives us the centroid of each obstalce. We utilize these information as follows:

- After the BuildWorldModel state, we sort the obstacles are sorted front-to-back. The planner then controls the behaviors such that the obstacles are passed in that order.
- The world coordinates allow to place the drone roughly in front of the obstacle before it starts to fly through it. We use the z-coordinate to set the drone altitude and the y-coordinate to decide whether to fly to the right or to the left after the drone passed an obstacle in order to see the next obstacle.

VII. Challenges

A. AR.Drone problems

The AR.Drone as device is quite unstable. Hoovering the drone at the same position for a couple of seconds is not possible since it drifts to one direction quickly despite there is a distinct function in the SDK for this task. The drone does not react to the sent commands very reliable – especially if it already moves in some direction. Developing software for the AR.Drone turned out to follow rather the try-and-error paradigm than basic software engineering principles. The documentation lacks on some important information for example regarding units of NavData values. The SDK is rather basic. As also mentioned in Section VII-B, navigation would be much easier if a FlyToPoint operation would be available for flying to a distinct point specified by absolute world coordinates.

B. `FlyToPoint` Behaviour

We put effort in developing an FlyToPoint behavior. Our approach uses the x- and y-speed measurements of NavData to estimate the current drone position in the world coordinate system. It then compares it to the target position and set pitch and roll according to this. The larger the offset, the bigger the angle. Unfortunately, our FlyToPoint behavior was too unreliable for integrating it in our program. We see two reasons for the disappointing result for this behavior.

- The x- and y-speed values in NavData are not very accurate enough and makes it hard to adjust roll and pitch properly.
- We implemented the behavior such that the x- and y-offsets of the drone and target position are linearly correlated with the pitch and roll values. However, this seems to be not feasible, because the speed of the drone increases with the time, even with constant orientation. We think that a linear dependency of the pitch and roll on the derivation of the x- and y- offsets would be more appropriate. However, we did not have enough time to implement such more sophisticated approach.

VIII. Conclusion and future work

We investigated the task of navigating through a 3D space with an aircraft. We concentrate on a particular scenario, in which the aircraft flies through a set of ring autonomously on a convinient trace. Our program on the AR.Drone managed to detect shapes and calculate depths quite reliable, but failed sometimes on the controlling and navigation part.

We feel that our program would run much more reliable on the new AR.Drone version, which was released after the project start. The new version brings greatly enhanced stability and the FlyToPoint operation, which we missed on different places above. Future work may use some of our proposed methods (like shape detection or depth calculation) in applications on new AR.Drone models.

References

[1] T. Krajník and J. Faigl and V. Vonásek and K. Košnar and M. Kulich and L. Přeučil, *Simple, Yet Stable Bearing-Only Navigation*, Journal of Field Robotics, Septemer 2010.

[2] C. Anderson, *Parrot AR.Drone Specification*, http://diydrones.com/profiles/blogs/parrot-ardrones-specs-arm9, 2010.

[3] S. Piskorski and N. Brulez and P. Eline *AR.Drone Developer Guide*, SDK 1.7, May 2011.

[4] C. Sanderson, *Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments* Technical report, NICTA, 2010.

[5] G. Bradski and A. Kaehler *Learning OpenCV: Computer vision with the OpenCV library* O'Reilly Media, 2008.