Real-Time Systems Course
Master Degree in Embedded Computing Systems

Catching Drone - 3D Simulation

Report

Student:
Gabriele Serra
gabriele_serra@hotmail.it
502651

Professor:
Prof. Giorgio Buttazzo

23/02/2017

"Once you have tasted flight, you will forever walk the earth with your eyes turned skyward, for there you have been, and there you will always long to return."

- Leonardo da Vinci

Preliminary information

The purpose of the assigned work was to design a program to control a virtual drone that catches a ball thrown by the user. In particular it was required the interaction with a 3D graphic simulation engine (Unreal Engine) to display a virtual environment where virtual drone operates. Drone modeling and control has to be implemented in Linux as real-time tasks.

Firstable, through a simple user panel, the user is able to change initial parameter of simulation. At this point simulation can start. The drone search for ball and through sensors try to predict the final position. A smart control drives drone to ball final position and, if possible, catch the ball.

At the end of simulation the program will restart in order to make another attempt. At any point simulation can be killed and restart from begin point with the R key.

Contents

List of Figures

List of Tables

1. Physical Models

1.1 Model of a drone

1.1.1 Introduction

A drone, or in our case a quadcopter, is a simple helicopter with four rotors usually arranged at the corners of a square body (thus with equal distance from the center of mass of the quadcopter). In this years drones has received more and more attention not only by researcher but also from people that uses quadcopters in everyday life for surveillance, search and rescue, record video, construction inspections and several other applications. The only way to control quadcopter trajector and position in space is to increase and decrease the angular speed of the rotors (or better operate on input of electric motors that spin rotors). A drone in space can fly forward/backward, up/down and left/right in three perpendicular axes. Also can changes its orientation through rotation about three perpendicular axes, often termed pitch, yaw, and roll. This freedom level of movement of a rigid body in three-dimensional space it's know as 6DOF[1].

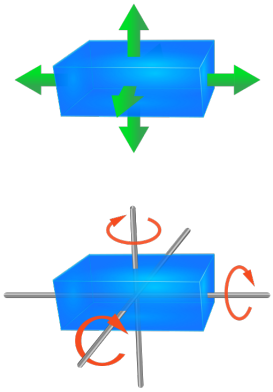


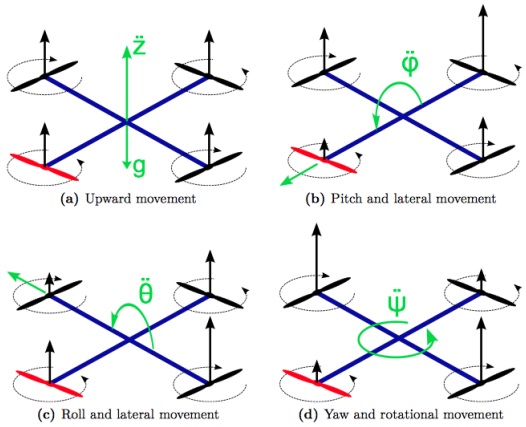
Figure 1.1: *Representation of freedom degrees of a rigid body*

[1] https://en.wikipedia.org/wiki/Six_degrees_of_freedom/

The resulting dynamics are nonlinear, especially after accounting for the aerodynamic effects such as drag and resistance to air.

1.1.2 Model

Each couple of rotor disposed on the same arm control the rotation of the quadcopter around an axis. In particular left and right rotor control the roll angle, back and front rotor control the pitch angle. Incrementing the thrust of a rotor and decrementing the thrust of the opposite one give us the possibility to maintain the stability of yaw angle and in the same time create a torque different from 0.



(a) Upward movement (b) Pitch and lateral movement

(c) Roll and lateral movement (d) Yaw and rotational movement

Figure 1.2: *Angular motion of a quadcopter*

In order to easily write drone mathematical model it's common to use different frame of reference: a body frame and an inertial fixed frame.[1]

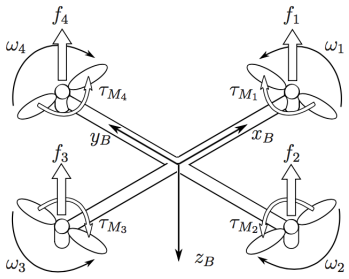


Figure 1.3: *Quadcopter body frame*

The position of drone in space is defined in the fixed inertial frame on x, y, z axes with $\boldsymbol{\xi}$. The angular position (attitude) of the body of drone is still defined in fixed frame with

three different angles with $\boldsymbol{\eta}$. The speed of the body instead are defined in body frame. In particular linear velocity are indicated with $\boldsymbol{V}$ and angular velocity are indicated with $\boldsymbol{A}$

$$\boldsymbol{\xi_F} = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{z} \end{bmatrix} \qquad \boldsymbol{\eta_F} = \begin{bmatrix} \boldsymbol{\phi} \\ \boldsymbol{\theta} \\ \boldsymbol{\psi} \end{bmatrix} \qquad \boldsymbol{V_B} = \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{v} \\ \boldsymbol{w} \end{bmatrix} \qquad \boldsymbol{A_B} = \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{q} \\ \boldsymbol{r} \end{bmatrix}$$

To pass from a frame to the other are needed rotation matrices.

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{C_\theta S_\psi} & \boldsymbol{C_\theta S_\psi} & \boldsymbol{-S_\theta} \\ \boldsymbol{S_\phi S_\theta C_\psi - C_\phi S_\psi} & \boldsymbol{C_\phi C_\psi + S_\phi S_\theta S_\psi} & \boldsymbol{S_\phi C_\theta} \\ \boldsymbol{C_\phi S_\theta C_\psi + S_\phi S_\psi} & \boldsymbol{S_\theta C_\phi S_\psi - S_\phi C_\psi} & \boldsymbol{C_\theta C_\phi} \end{bmatrix} \qquad \boldsymbol{W} = \begin{bmatrix} \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{-S_\theta} \\ \boldsymbol{0} & \boldsymbol{C_\phi} & \boldsymbol{C_\theta S_\phi} \\ \boldsymbol{0} & \boldsymbol{-S_\phi} & \boldsymbol{C_\theta C_\phi} \end{bmatrix}$$

Thus we can relate the two frame:

$$\boldsymbol{\dot{\eta_F} = W^{-}1 \cdot A_B} \qquad \boldsymbol{\dot{\xi_F} = R^{-}1 \cdot A_B}$$

$$\boldsymbol{A_B = W \cdot \dot{\eta_F}} \qquad \boldsymbol{V_B = R \cdot \dot{\xi_F}}$$

The angular velocity of each rotor identified with $\boldsymbol{\omega_i}$, generate a force $\boldsymbol{f_i}$ in the direction of the rotor axis.

$$\boldsymbol{f_i = d \cdot \omega_i}$$

In mechanical engineering, force orthogonal to the main load is referred to as thrust. The combined forces of rotors create thrust T in the direction of the body z-axis.

$$\boldsymbol{T = F_t = f_f + f_b + f_r + f_l = d \cdot \sum_i \omega_i^2}$$

Furthermore angular velocity with angular acceleration generate a torque $\boldsymbol{\tau_i}$ around the axis of the rotor.

$$\boldsymbol{\tau_i = k \cdot f_i + I_M \cdot \dot{\omega}_i}$$

Where k is a constant for the conversion with a typical value of 0.08. In steady state flight $\boldsymbol{\dot{\omega}}$ it's almost null, thus it can neglected. With this information we can describe the torque in the direction of each body frame angles

$$\boldsymbol{\tau_\phi = L \cdot (f_l - f_r)}$$

$$\boldsymbol{\tau_\theta = L \cdot (f_f - f_b)}$$

$$\boldsymbol{\tau_\psi = (\tau_r + \tau_l - \tau_f - \tau_b) = k f_r + k f_l - k f_f - k f_b}$$

Hence it's possible to derive a relation to obtain the force to be applied on each motor:

$$\begin{bmatrix} f_t \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -L & 0 & -L \\ -L & 0 & L & 0 \\ -k & k & -k & k \end{bmatrix} \begin{bmatrix} f_f \\ f_r \\ f_t \\ f_b \\ f_l \end{bmatrix} = M \begin{bmatrix} f_f \\ f_r \\ f_t \\ f_b \\ f_l \end{bmatrix}$$

$$\begin{bmatrix} f_f \\ f_r \\ f_b \\ f_l \end{bmatrix} = M^-1 \begin{bmatrix} f_t \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & 0 & \frac{-1}{2L} & \frac{1}{4k} \\ \frac{1}{4} & \frac{-1}{2L} & 0 & \frac{1}{4k} \\ \frac{1}{4} & 0 & \frac{1}{2L} & \frac{-1}{4k} \\ \frac{1}{4} & \frac{1}{2L} & 0 & \frac{1}{4k} \end{bmatrix} \begin{bmatrix} f_t \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix}$$

The matrix M^1 is called Force Allocation matrix.

Kinematics

To figured out how the drone can move in space we can start from Newton-Euler equation. In particular we can start using the second law of Newton

$$F = ma$$

$$F = m \cdot \ddot{\xi}_F$$

In the inertial frame, the centrifugal force is null thus the acceleration is due to thrust, gravity, and linear friction. In our simplified model linear friction of air is not considered, so we can write:

$$\ddot{\xi}_F = \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} + \frac{1}{m} \cdot R \begin{bmatrix} 0 \\ 0 \\ -f_t \end{bmatrix}$$

The thrust must be multiplied by R to change the reference frame. Regarding the angular acceleration, we derive the rotational equations of motion from Euler's equations. In the body frame, the angular acceleration of the inertia $I\dot{A_B}$ the centripetal forces $V_B \times (IA_B)$ are equal to the external torque τ

$$I\dot{A_B} + A_B \times (IA_B) = \tau$$

If we explicit $\dot{A_B}$:

$$\dot{A_B} = \begin{bmatrix} \tau_\phi I_x^- 1 \\ \tau_\theta I_y^- 1 \\ \tau_\psi I_z^- 1 \end{bmatrix} - \begin{bmatrix} \frac{I_y - I_z}{I_x} qr \\ \frac{I_z - I_x}{I_y} pr \\ \frac{I_x - I_y}{I_z} pq \end{bmatrix}$$

1.1.3 Implementation of state update

The physical model just derived bring directly to the implementation of function to update the quadcopter state in space. In the state structure of drone I've choose to keep the four vector already seen $\boldsymbol{\xi_F}, \boldsymbol{\eta_F}, \boldsymbol{V_B}, \boldsymbol{A_B}$ plus the $\dot{\boldsymbol{\xi_F}}, \dot{\boldsymbol{\eta_F}}$ to avoid the use of derivator. Also another vector kept in state structure is the **rotor duty cycle** values. In fact electrical motors work with a 0-1 interval value that indicates the power from 0 to Max force of the rotor. Hence with all this information implement an algorithm to update the quadcopter state in time is trivial. The starter point is to evaluate the quadcopter angular and linear acceleration using the knowledge of thrust and torques acting on body of quadcopter. Once obtained the acceleration on quadcopter, the biggest work is done because it's only needed to update all the state information.

1.1.4 Control design

The mathematical model we just obtained is clearly non-linear. We have to do some simplification in order to design a control system. First in steady state we can suppose that $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ are small.[2] In that case the relation between angular velocity in body and fixed frame became

$$\begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} = \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{q} \\ \boldsymbol{r} \end{pmatrix} \qquad \dot{\boldsymbol{\eta_F}} = \boldsymbol{A_B}$$

The factor due to coriolis force it's almost null and can be neglected. Thus we can write

$$\dot{\boldsymbol{A_B}} = \begin{bmatrix} \tau_\phi \boldsymbol{I_x^-1} \\ \tau_\theta \boldsymbol{I_y^-1} \\ \tau_\psi \boldsymbol{I_z^-1} \end{bmatrix}$$

Combining both the previous simplification we can calculate linear the acceleration and torques.

$$\ddot{\boldsymbol{\eta_F}} = \begin{bmatrix} \boldsymbol{I_x^-1} \\ \boldsymbol{I_y^-1} \\ \boldsymbol{I_z^-1} \end{bmatrix} \cdot \tau \qquad \tau = \boldsymbol{I} \cdot \ddot{\boldsymbol{\eta_F}}$$

Firstable in order to design the control of quadcopter we have to design the stability. The aim is to stabilize and command using as control variables

- Roll angle $\boldsymbol{\phi}$
- Pitch angle $\boldsymbol{\theta}$

- Yaw angle $\boldsymbol{\psi}$

We use the simplified model in which:

$$\ddot{\boldsymbol{\phi}} = \frac{\boldsymbol{\tau_\phi}}{\boldsymbol{I_x}} \qquad \ddot{\boldsymbol{\theta}} = \frac{\boldsymbol{\tau_\theta}}{\boldsymbol{I_y}} \qquad \ddot{\boldsymbol{\psi}} = \frac{\boldsymbol{\tau_\psi}}{\boldsymbol{I_z}}$$

We can write the transfer function as

$$\boldsymbol{G_\phi(s)} = \frac{\boldsymbol{\phi(s)}}{\boldsymbol{\tau_\phi(s)}}$$

$$\boldsymbol{s^2\phi(s)} = \frac{\boldsymbol{\tau_\phi(s)}}{\boldsymbol{I_x}}$$

$$\boldsymbol{\phi(s)} = \frac{\boldsymbol{\tau_\phi(s)}}{\boldsymbol{I_x s^2}}$$

And basically it's the same for pitch and yaw angle.

The second objective of this simulation is the trajectory control. The main idea is based on the fact that angular and linear acceleration of drone are calculated starting from rotor force (and consequently by rotor duty cycle). Thus if we can relate the position of drone to linear acceleration it's simple to derive a trajectory controller. $\boldsymbol{\xi_F}$ is our linear position vector. $\ddot{\boldsymbol{\xi}}_{\boldsymbol{F}}$ is the linear acceleration vector. So the relation between acceleration and position as we know is purely derivative. Follow this way of reasoning we can write the transfer function in the same way as before:

$$\boldsymbol{x_F(s)} = \frac{\ddot{\boldsymbol{x}}_{\boldsymbol{F}}\boldsymbol{(s)}}{\boldsymbol{s^2}}$$

$$\boldsymbol{y_F(s)} = \frac{\ddot{\boldsymbol{y}}_{\boldsymbol{F}}\boldsymbol{(s)}}{\boldsymbol{s^2}}$$

$$\boldsymbol{z_F(s)} = \frac{\ddot{\boldsymbol{z}}_{\boldsymbol{F}}\boldsymbol{(s)}}{\boldsymbol{s^2}}$$

In this way we can control position starting from acceleration. But we still miss something, we have to relate desired acceleration to rotor duty cycle that is the only thing we can control. This is possible using the control of stability imposing angles and thrust desired. Using the equations write in the previous section and the simplification we have already seen, we can to explicit roll, pitch, yaw and thrust. Actually yaw will be set to 0 because we can use only pitch and roll to control acceleration.

$$\boldsymbol{\phi} = \mathbf{arctan}\,(\frac{\ddot{\boldsymbol{y}}_{\boldsymbol{F}}\boldsymbol{(s)}}{\boldsymbol{g} + \ddot{\boldsymbol{z}}_{\boldsymbol{F}}\boldsymbol{(s)}})$$

$$\theta = \arctan\left(\frac{\ddot{x}_F(s)}{g + \ddot{z}_F(s)}\right)$$

$$T = \frac{mg}{C_{\ddot{x}_F(s)} \cdot C_{\ddot{y}_F(s)}} + \ddot{z}_F(s) \cdot C_{\ddot{x}_F(s)} \cdot C_{\ddot{y}_F(s)}$$

This results can be passed to stability control to actuate required angles and thrust. To clarify our control task a simple picture can give an idea on how state vector are related.[3]

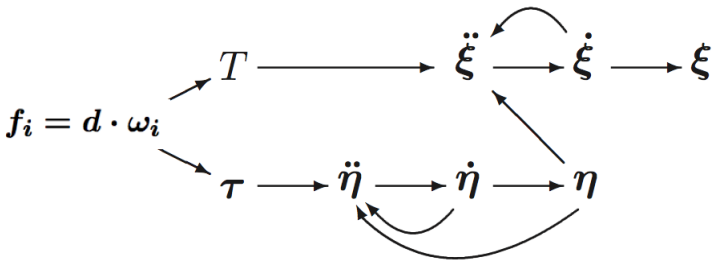


Figure 1.4: *Relation between state vector of drone*

1.1.5 Implementation of control

The controllers used for this simulation are PD's (proportional derivative controller). This kind of controller are very common in application of this type. The PD's controller are very simple, in general a PD controller have a component proportional to the error (desired value - observed value) and a component proportional to the derivative of the error. In general a PID control can be written as

$$e(t) = x_d(t) - x(t)$$

$$u(t) = K_P \cdot e(t) + K_D \cdot \frac{de(t)}{dt} + \int_0^t e(\tau)d\tau$$

In our case we throw off the integral piece to obtain a PD. Let's start from stability control, we focus only on roll angle but in general it's the same for pitch and yaw.

$$PD_{out} = \tau_\phi = K_P(\phi_d - \phi) + K_D(\dot{\phi_d} - \dot{\phi})$$

But, at steady state, we don't want angular velocity. Hence our formula become

$$PD_{out} = \tau_\phi = K_P(\phi_d - \phi) - K_D\dot{\phi}$$

The same ideas are used for trajectory control. Skipping some math we can conclude

$$PD_{out} = \ddot{x} = K_P(x_d - x) - K_D\dot{x}$$

Actually we can calculate gains using MATLAB or experimental results. Also our input should be bounded because our system is BIBO stable (bounded input - bounded output). If our input cannot be bounded, is needed a saturation to limit output of PD's. All this relation holds for the S plane but since we work in time discrete domain it is possible to bring all time continue relation to time time discrete relation with a transformation. Actually all this

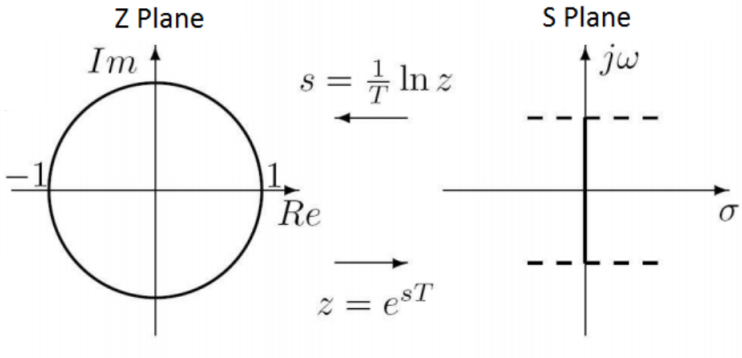


Figure 1.5: *Relation between S and Z plans*

calculation will be left to MATLAB.[4]

1.2 Model of the ball

The motion of the ball is modeled with a linear motion and in particular a non uniform linear motion. The acceleration of the ball (or better deceleration) it's equal to gravitational acceleration. If we denote with $\boldsymbol{V}$ the vector of linear speed of the ball in space and with $\boldsymbol{\xi}$ the position, we can write

$$\boldsymbol{V} = \begin{bmatrix} \boldsymbol{u_0} \\ \boldsymbol{v_0} \\ \boldsymbol{w_0} \end{bmatrix} - \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{g} \end{bmatrix} \boldsymbol{t}$$

$$\boldsymbol{\xi} = -\frac{\boldsymbol{1}}{\boldsymbol{2}} \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{g} \end{bmatrix} \boldsymbol{t^2} + \begin{bmatrix} \boldsymbol{u_0} \\ \boldsymbol{v_0} \\ \boldsymbol{w_0} \end{bmatrix} \boldsymbol{t} + \begin{bmatrix} \boldsymbol{x_0} \\ \boldsymbol{y_0} \\ \boldsymbol{z_0} \end{bmatrix} \boldsymbol{t}$$

The ball is thrown by the user through a little cannon. For the sake of simplicity the force applied by the cannon is not modeled as an impulse. In that case it's needed to calculate the derivative of the momentum.

$$\boldsymbol{F} = \frac{\boldsymbol{dp}}{\boldsymbol{dt}} \qquad \boldsymbol{p = mv}$$

The force applied to the ball will cause an acceleration different from 0 that produce an update of velocity. The same behavior (and without take care about damping of acceleration in time and so on) can be obtained choosing an initial speed different from 0. Thus the user can regulate the power of the launch.

1.3 Ball position estimate

Another important part of simulation is the estimation of position of ball. To make simulation closest to real we start from the assumption that drone don't know almost anything about ball but he have some sensors to observe data that he need. Hence the idea is to use a proximity sensor (a very powerful one, probably doesn't exist!) or a camera to calculate ball position in space and ball velocity. So our position estimation algorithm start with ball actual velocity and ball actual position.

Using the kinematics law we can calculate the actual flight time $\boldsymbol{t}$ of the ball.

$$\boldsymbol{t_a = \frac{\dot{z} - \sqrt{\dot{z}^2 + 2gz}}{-g}}$$

With the flight time, actual position and actual velocity we can calculate simply the initial position of the ball namely the point in which user has decided that ball must start.

$$\boldsymbol{x_i = x - \dot{x} \cdot t_a}$$

$$\boldsymbol{y_i = y - \dot{y} \cdot t_a}$$

In the same way we can calculate the initial velocity of ball in Z axis

$$\boldsymbol{\dot{z}_i = \dot{z} + g \cdot t_a}$$

At this point if we imposed a final Z position of the ball we can use the ballistic equation to calculate the total time of flight and the final position in X and Y.

$$\boldsymbol{z_f = z_0}$$

$$\boldsymbol{t_f = \frac{\dot{z}_i + \sqrt{\dot{z}_i^2 - 2g \cdot z_f}}{g}}$$

$$\boldsymbol{x_f = t_f \cdot \dot{x} + x_i}$$

$$\boldsymbol{y_f = t_f \cdot \dot{y} + y_i}$$

In this way with a simple camera or a proximity sensors we have calculated the information we need.

2. Design choice

Several design choices has been taken in order to simulate correctly the environment.

2.1 Drone attitude data and sensors

One on the most interesting key point of simulation is the drone attitude estimation. Actual attitude is needed every instant of time by the controller/driver of the drone to stabilize and control trajectory of the body of quadcopter. Until some years ago wasn't so easy to estimate with precision position and speed of a body. Nowadays on most of high end drone are present accelerometer, gyroscope and magnetometer implementing a real attitude and heading reference system.[1] In our simulation we simply imagine to have this system of sensors mounted on our quadcopter body.

2.2 Simulation state

In order to well structure the simulator a set of internal state have been mapped. Clearly task interaction depends by the current state. Thus the idea is that simulation always been in one of the follow state:

- STOPPED
- RUNNING
- PAUSED

Briefly we can associate to each state the simulation behavior: STOPPED means that the simulation is not running (the application is just opened or simulation is re-setted); RUNNING means that the quadcopter had found the ball and he is trying to catch the ball; PAUSED means the simulator is in pause, so physical environment is frozen. The user with keyboard

[1] https://en.wikipedia.org/wiki/AHRS

decide to change the simulation state. To focus on each action required by each transition, it's needed a transition table. The transition table map a function of type

$$state_{n+1} = f(state_n, inputFromPanel)$$

The required action will be executed by a supervisor task.

Next state Current state	STOPPED	RUNNING	PAUSED
STOPPED	**NO-INPUT** drone/ball init UDP task-start	**ENTER** DRN task-start BLL task-start DRV task-start	ERR
RUNNING	**R** ALL task-stop drone/ball reset	**NO-INPUT** -	**BACKSPACE** ALL task-stop
PAUSED	**R** ALL task-stop drone/ball reset	**ENTER** ALL task-start drone/ball reset	**NO-INPUT** -

Table 2.1: *State transition table*

In table the follow notation has been used: with DRN is identified the drone physical task, with BLL the ball physical task, with DRV the controller mounted on drone task and with UDP the udp packet sender (see next chapter), with a '-' is marked a nothing-to-do action and with ERR and impossible (and erroneous) transition.

2.3 Simulation world and ball parameter

2.3.1 World

Another important design choice regard the world in which simulation take part. My idea is to limit the 3d real world in a square. The top left corner is situated at the coordinates (100, 100) and the bottom right corner is situated at the coordinates (-100, -100). Hence the length of each side of the square is 200 meters. This means the world measure 4 hectares (ha). The same thing is done for the 2d graphic world in which user can position ball and drone. This approach brings pros and cons. The most important advantage of limit the world is that we

are sure ball and drone input are always bounded. The most important disadvantage is that we need a function to map the 2D world coordinates in 3D world coordinates.

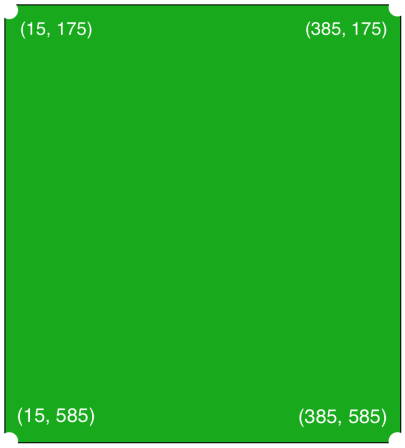


Figure 2.1: *A screenshot of entire 2D simulation world*

This problem is resolved with a simple function that maps a coordinate from a range of input into a coordinate in a range of output.

2.3.2 Ball parameter

The ball, when thrown by the user, has got two different parameters, power and direction. The power depends from Y coordinate of draggable ball in user panel. I've choose to limit the power in 10-unit-scale from 0,0 to 10,0 and add a block in "power bar" every integer unit of power. The direction is still limited in 10-unit scale but is mapped from -5,0 to 5,0. The direction meaning is "towards the centre of the map", namely if direction is not modified the ball go towards the map centre, else the trajectory is modified. For sake of clarity, if the ball is positioned at the bottom right corner and the user choose all right direction, the ball moves towards the top left corner. If choose all left direction, the ball move towards the bottom left corner. If direction is leaved as default the ball moves towards the centre of map. This is implemented with 3-step process. First step is to calculate the distance between ball and center of map. The second step is to calculate the angle from 0 to the point with the arctangent of quotient of y and x. Once calculated the angle of point, the third step is to use **sin** and **cos** to map the weight of right and left to X and Y velocity.

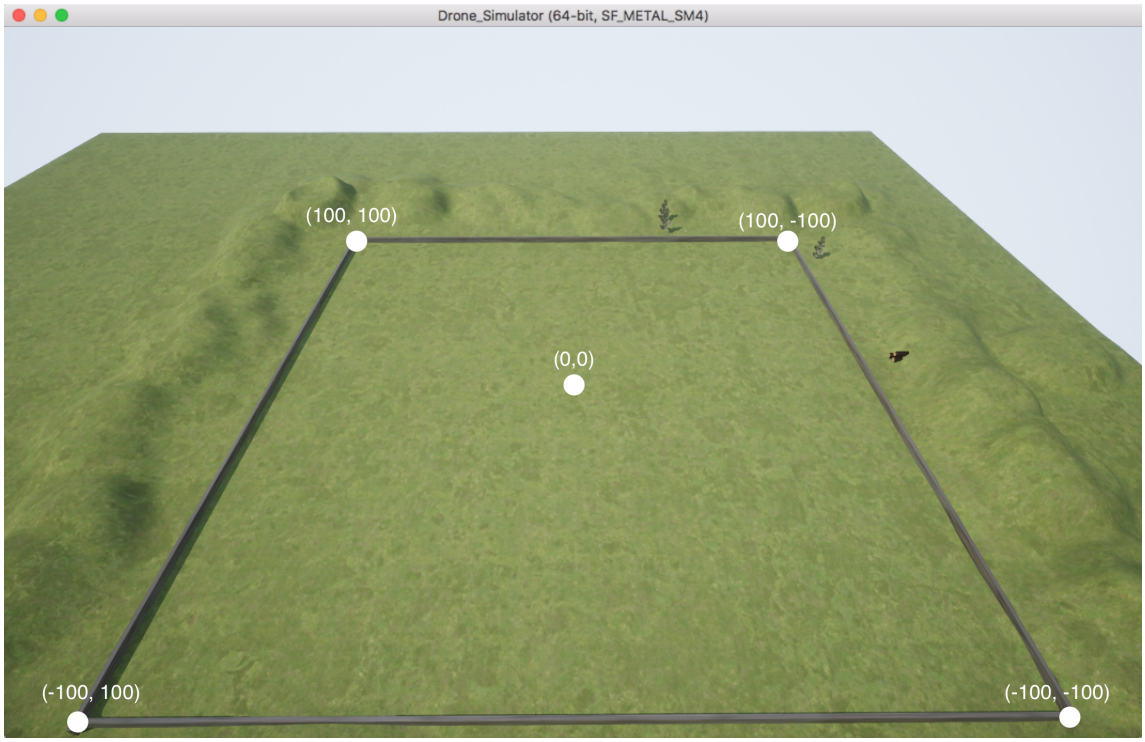


Figure 2.2: *A screenshot of entire 3D simulation world*

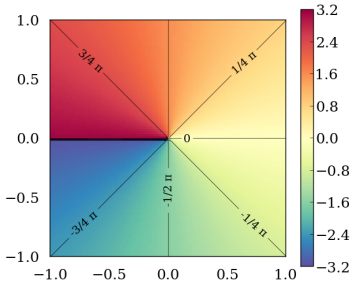


Figure 2.3: *Arctangent of quotient of y and x*

3. User interface

3.1 Introduction

The user interface of the simulation is mainly composed by two different entities, user panel and 3d simulator viewer. User panel and 3d simulator viewer exchange information among them using UDP packets. The user panel it's an interface realized using Allegro Lib, running on Linux in VGA (8 bit) mode. The 3d simulator viewer it's an MacOS/Windows application realized using Unreal Engine 4 Engine. The highest level user experience is reached using two different devices, one running Linux and one running Windows or MacOs. In order to run the 3D viewer simulation there are some hardware recommended requisites, that are listed in table below. Naturally more updated OS or better hardware specification increase the usability of the application.

Platform	**OS Version**	**Processor**	**Memory**	**Video Card**
Windows	Windows 7 64bit	2.5 GHz	8 GB RAM	DirectX 11 compatible
Mac	Mac OS X 10.10	2.5 GHz	8 GB RAM	OpenGL 4.1 compatible

Table 3.1: *Recommended hardware specifications*

3.2 User panel

The user panel allow the user to control the state of simulation and to change some parameters about initial position, ball power and so on. The interface is clearly simple and explain how to use keyboard and mouse to change settings. An image of the user panel can be seen below. The interface is thought as three boxes, each with a different role.

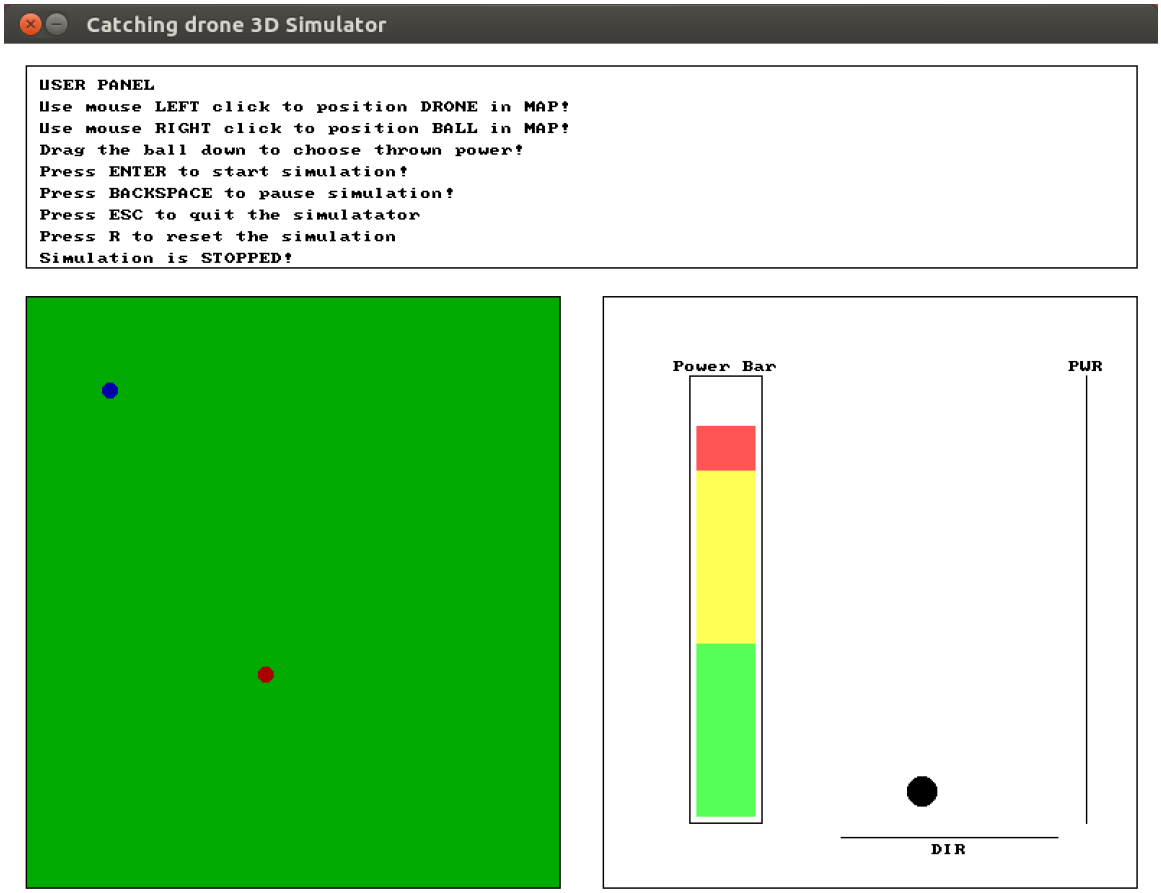


Figure 3.1: *A screenshot of user panel interface*

3.2.1 Three boxes interface

Top box

The top box explain to the user how the interface can be used with keyboard and mouse. Furthermore the state of simulation is printed on screen.

Map box

The map box allows the user to position drone and ball in world and to follow the progress of simulation. In fact, while simulation is running, ball and drone circle position are updated.

Power box

The power box has the meaning of a "sling". The user can drag the ball to the bottom of screen to increase the power of the throw and change the direction. Ball and drone **cannot**

go out of map box.

3.2.2 User possible action

The user can do some action to control the simulation. At first, when the application is open, map box and power box are empty. User can click with the left button of mouse on map box to choose the initial position of the drone (red circle). In simulation viewer the drone change it's position. The same thing can be done with the right mouse button to choose the ball (blue circle) initial position. Using the power box user can choose power and direction as explained. With the keyboard button, simulation can be started (ENTER), paused (BACKSPACE) or reset (R). Pressing ESC, the application close itself.

3.3 3d simulator viewer

The 3d simulator viewer can be used by the user to follow the simulation in a 3D world. Once opened, the only interaction is limited to change of view. The user in fact can use the mouse to change perspective and mouse left/right click to zoom-in/zoom-out the camera mounted on drone. To control the simulation user must open the user panel.

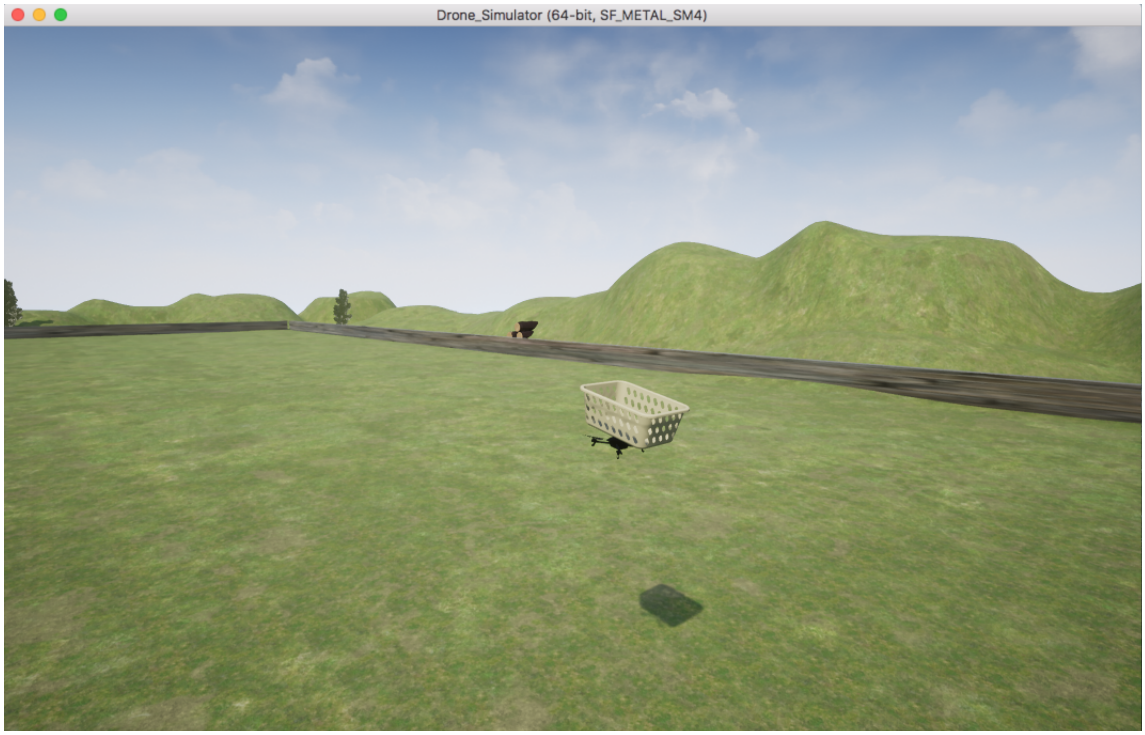


Figure 3.2: *A screenshot of drone searching for ball*

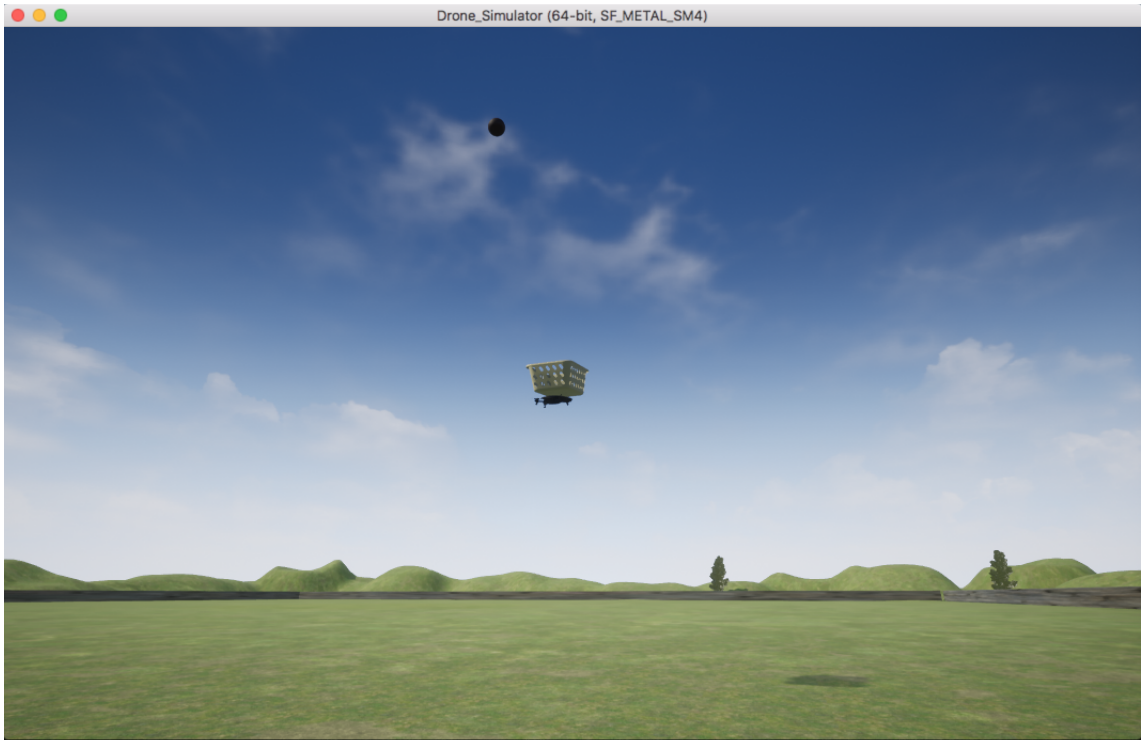


Figure 3.3: *A screenshot of drone catching the ball*

4. Tasks and data structures

4.1 Different tasks

The task of simulate an environment involves some concurrent physical and mathematical process. Six tasks has been identified in this works.

- Udp graphic packet sender (UDP)
- Drone driver controller logic (DRV)
- Drone physics evolution (DRN)
- Ball physics evolution (BLL)
- User panel handler (PNL)
- Simulation supervisor (SPV)

4.1.1 Udp graphic packet sender

This task works as a substitute for graphics display of the simulation. In fact for handle a 3D environment has been used Unreal Engine engine. The udp sender sends a packet with the graphic information needed by the engine (ball and drone position) every time. Unreal Engine will do the rest.

4.1.2 Drone driver controller logic

The drone stability controller logic will simulate the microcontroller mounted up on the quadcopter body. The goal of the controller is to check information from sensors and drive the quadcopter to catch the ball. Clearly stability of quadcopter is needed and handled by it.

4.1.3 Ball and drone physics evolution

The physics of the environment is fundamental to simulate the movement of drone and ball. Hence these threads take care to evolve, following the law of physics, the trajectory, position, speed, acceleration and so on.

4.1.4 User panel handler

The user that run the software need to change some parameters in order to make different simulation. Also he may need to stop or re-run the simulation. Every interaction with user are done by User panel handler thread.

4.1.5 Simulation supervisor

The simulation supervisor thread take care of changing the state of simulation based on user input that arrives through user panel, start ball and drone threads and udp sender in the right way and right on time.

4.2 Data structures

In a system with threads that exchange information among them are needed some data structures. In this perspective four data structures have been identified.

- dstate (Drone)
- bstate (Ball)
- pstate (Panel)
- cstate (Controller)

Each of data structures contains the state of respective object. The structure **dtsate** contains physical quantities about the state of drone in the environment such as linear and angular velocity, linear and angular position in space and so on. The same is for ball, using **bstate** structure. The controller of drone act only on drone rotor duty cicle thus **cstate** contains only the drone's rotor duty cicle. The structure **pstate** contains all information about user graphic panel such as the state of simulation imposed by the user, the drone and ball position on map and so on.

4.3 Interactions among tasks and data structures

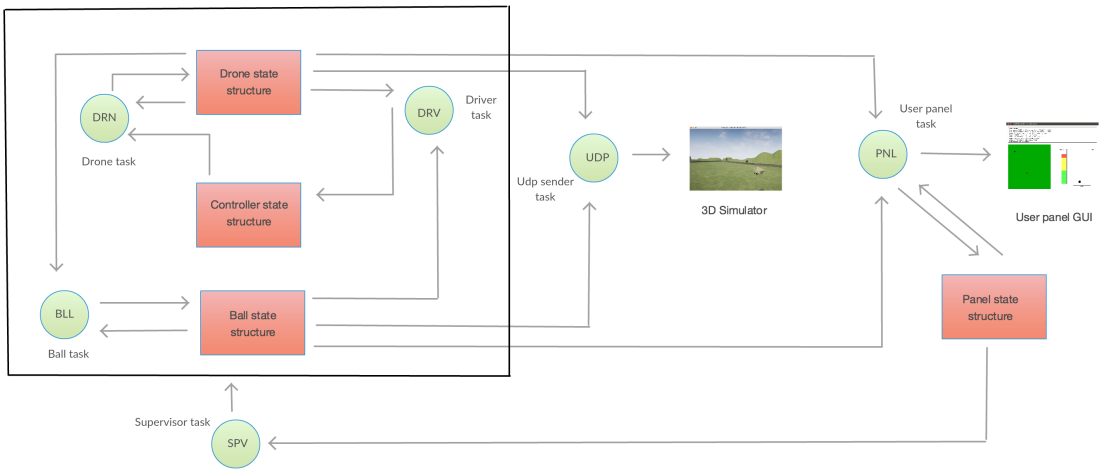


Figure 4.1: *Interactions among tasks and data structures*

4.4 Timing and schedule

In this work six different thread that executes (virtually) in the same time. In an uni-processor machine clearly threads are executes one by one thus it's needed a proper schedule algorithm. I started from simple assumption to define the constraint tasks have to respect. Videogames, film, animation and so on, need an high rate of frame update to hide the observer eyes differences between one frame and another. Some studies and observation has discovered that human eyes works at about 25-30 frame per second. This information means that if the video is updated at least with 30 frame per second, the observer detect the update as a movement. Hence the start assumption is that we want at least 30 graphic update per second. $\mathbf{30}\boldsymbol{fps} = \frac{\mathbf{1}}{\mathbf{30}}\boldsymbol{second - per - frame} \approx \mathbf{0.033}$ This means we have to send at least a graphic update every 33 ms and this means Udp sender task and User panel task have to be executed at least once in a 33ms period. Actually would be useless send a graphic update without new data to display! This way of reasoning bring us to say that Drone and Ball physical evolving tasks have to be executed at the same rate of graphical updates. Controller tasks instead depends from hardware that we want to simulate. For sake of simplicity we can say that the slowest hardware device (the controller logic, the gyroscope, the camera or every other device mounted on controller) send data at $\mathbf{50}\boldsymbol{hz} = \mathbf{20}\boldsymbol{ms}$. The least important thread is the supervisor. Supervisor task don't need too much time to react at user action.

Also if user intention to change state is not executed in time it's not a problem of graphics. Thus we can assign the slowest period and the lowest priority.

4.4.1 Priority assignment

The same way of reasoning applied to calculate the period of each task is used to assign priority. Without nothing to send it's useless to send graphic update! Hence we assign the highest priority to the task that managing physical evolution and to the controller. Highest priority is assigned also to udp sender because user eyes are really sensible 3D graphic evolution. Medium priority is assigned to task that manage user panel. The lowest priority is assigned to the supervisor task.

4.4.2 Scheduling

With all information collected we can derive that we need a Real-Time schedule algorithm that ensure respect of deadline, and order of execution. This way of reasoning drive to use the SCHED_FIFO linux kernel scheduling. With FIFO we can sure thread with same priority are managed FIFO. At the end of every execution, thread can auto-suspend to leave the remaining time to others. For simplicity deadline and period are equal.

The parameter derived are:

	Period	Priority
DRV	20 ms	2
DRN	30 ms	3
BLL	30 ms	3
UDP	30 ms	3
PNL	30 ms	2
SPV	50 ms	1

Table 4.1: *Period and priority of tasks*

5. Result and future work

Some simulations are been used to "tune" various parameter of drone and ball.

5.1 Scale factor

To make simulation work properly we need some "scale factor". In fact we need to multiply/attenuate some physical quantities to give a physical sense. The scales introduced are regarding the ball.

5.1.1 User to physical quantities

The user-related quantities (power and direction) are mapped in a 10-unit scale. But how a unit of power or direction is reflected on X,Y ball initial velocity? The effect are described in relation of increasing the constant. The opposite effect is related to decreasing of the constant. The **BLDIRSCALE** constant re-map direction in a -BLDIRSCALE/+BLDIRSCALE scale, namely attenuate the scale of direction. The **BLVELSCALEX** is used as multiplier to increment the effect of user direction choose on X initial velocity of ball. The same thing is for **BLVELSCALEY** constant. The action of power on Z-initial velocity of ball is regulated by **BLVELSCALEZ** constant. Also simulation result suggest that is needed another scale factor for gravity deceleration of ball. Our model doesn't take care of drag of air factor thus the ball fall with an unbounded speed. The introduction of **BLACCSCALEZ** should help attenuating the force of gravity that act on the ball.

5.2 Capacity of drone to catch the ball

The last topic is result chapter is dedicated to the real objective of this work: drone is capable to catch the ball in time? This topic maybe require some and some paragraph but briefly the answer is "sometimes". The main problem when drone doesn't catch the ball is due to the time response of drone controller. In fact overshoot and high settling time often cause drone reach a

really close point but with some seconds of lateness. Another big problem is the linearization of system. The assumption are built on the fact that pitch and roll angles are close to 0 and this is not always true. Future works surely must focus on different controller of quadcopter (PID instead PD or different approach with pole placement and filter of disturbance) and to a little more accurate model of physical environment.

Bibliography

[1] G. Pugliese Carratelli and M. Del Duca, "Controllo di un quadcopter," 2012.

[2] W. R. Beard, "Quadrotor dynamics and control." Brigham Young University, 2008.

[3] T. Luukkonen, "Modelling and control of quadcopter." Aalto University, 2011.

[4] D. Casini, "Quadrotor control with smartphone," 2015.