Simulator of a drone ball follower

Project for the exam of Real Time System of prof. Buttazzo

Dott. Ing. Borgioli Niccolò

Sommario

Goal of the project

The project aims to realize a real time simulator of a drone that have to follow a ball. The ball movement is generated randomly by a dedicated task. The simulator has to recognize and follow the ball using a virtual camera mounted on the drone.

Specifications

The visualization of the environment should be done in Unreal Engine, while all of the computation regarding image recognition, trajectory planning, drone control and system simulation have to be done by a separated program running on Linux using Allegro 3 Library.

Physical models

Drone model

The drone we are going to simulate is a quadcopter: a multirotor helicopter that is lifted and propelled by four rotors, it can be easily describe as a rigid body with four forces applied at his corners. The drone can move along three axes (x, y and z) that allows the vehicle to move up/down, forward/backward and left/right, moreover can change his orientation by rotating along three perpendicular axes attached to his body and these rotations are called: roll (rotation around x axis), pitch (rotation around y axis) and yaw (rotation around z axis). Thus the rigid body has 6 degrees of freedom (DOF) in the three dimensional space.

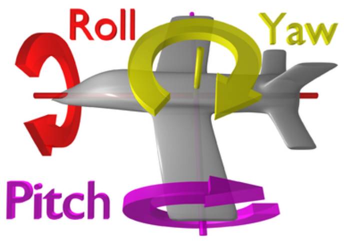


Figure 1 - Roll, pitch and yaw movement. (Wikipedia)

The drone dynamics is controlled by increasing or decreasing the rotational speed of the rotors:

- each couple of rotors on the same arm rotates in the same direction in order to control the yaw of the vehicle; by reducing or increasing the speed of a couple is possible to adjust yaw.
- by increasing thrust on a rotor and reducing on the opposite one is possible to adjust roll or pitch.
- increasing or decreasing the speed of all rotors the drone hoovers on his position or adjusts his altitude.

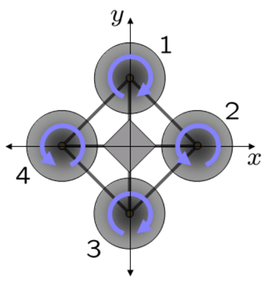


Figure 2 - rotors rotation (Wikipedia)

In order to mathematically describe the model of the drone is useful to define two different reference frames: the drone fixed frame (that is attached to the drone rigid body) and the inertia fixed frame.

In the space, the position of the drone is described respect to the inertia fixed frame on the x, y and z axes and his orientation (attitude) is described still respect to the inertia fixed frame with three different angles: roll (ϕ), pitch (θ) and yaw (ψ).

Moreover, the speed of the body is described into the body fixed frame: linear velocity are indicated with u, v and w, while the angular ones by p, q and r.

$$r_f = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \Omega_f = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \quad V_b = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \quad A_b = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

To move from a frame to the other are needed two rotation matrixes: R (for the position) and W (for the angles).

$$R = \begin{bmatrix} \cos(\phi) * \sin(\psi) & \cos(\theta) * \sin(\psi) & -\sin(\theta) \\ \cos(\psi) * \sin(\phi) * \sin(\theta) - \cos(\phi) * \sin(\psi) & \cos(\phi) * \cos(\psi) + \sin(\phi) * \sin(\theta) * \sin(\psi) & \sin(\phi) * \cos(\theta) \\ \cos(\phi) * \sin(\theta) * \cos(\psi) + \sin(\phi) * \sin(\psi) & \sin(\theta) * \cos(\phi) * \sin(\psi) - \sin(\phi) * \cos(\psi) & \cos(\theta) * \cos(\phi) \end{bmatrix}$$

$$W = \begin{bmatrix} 1 & 0 & -\sin(\theta) \\ 0 & \cos(\phi) & \cos(\theta) * \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\theta) * \cos(\phi) \end{bmatrix}$$

Where:

$$\dot{\Omega}_f = W * A_b \qquad \dot{r_f} = R * V_b$$

Each rotor rotates at a velocity w_i and generates a force $f_i = b * w_i^2$ perpendicular to itself and a reactive torque due to the rotation $Q_i = k * f_i$ around the axis of the rotor.

In the drone fixed frame is called thrust the total force perpendicular to the x/y plane. This force T is directed on the z axis of the drone fixed frame:

$$T = \sum_{i=1}^{4} f_i = \sum_{i=1}^{4} b * w_i^2 = b * \sum_{i=1}^{4} w_i^2$$

Moreover the torque in each direction of the drone frame can be described:

$$\tau = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} l * f_4 - l * f_2 \\ l * f_3 - l * f_1 \\ k * (w_2^2 + w_4^2 - w_1^2 - w_3^2) \end{bmatrix}$$

Where l is the length of the arm of the drone (in the model all of the arms of the drone are considered equals).

Putting thrust and torque together I can write in matrix form:

$$\begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} b & b & b & b \\ 0 & -l*b & 0 & l*b \\ -l*b & 0 & l*b & 0 \\ -k & k & -k & k \end{bmatrix} * \begin{bmatrix} w_1^2 \\ w_2^2 \\ w_3^2 \\ w_4^2 \end{bmatrix} = M * \begin{bmatrix} w_1^2 \\ w_2^2 \\ w_3^2 \\ w_4^2 \end{bmatrix}$$

Thus we have defined the force allocation matrix M that gives us a relation between the rotors speeds and the forces we are dealing with.

Consider now the Newton-Euler equations:

In the inertial frame the acceleration is due to only to gravity and thrust, so for the first Newton-Euler equation:

$$\ddot{r_f} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = - \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} + \frac{1}{m} * R * \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix}$$

Consider now the angular acceleration: the external torque τ is equal to the centripetal forces $A_b \times (I * A_b)$ and the angular acceleration of the inertia $I * \dot{A}_b$ in the drone fixed frame:

$$\tau = I * \dot{A}_b + A_b \times (I * A_b)$$

So from this we can obtain $\dot{A}_b$:

$$\dot{A}_b = \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \tau_\phi * I_x^{-1} \\ \tau_\theta * I_y^{-1} \\ \tau_\psi * I_z^{-1} \end{bmatrix} - \begin{bmatrix} \frac{I_y - I_z}{I_x} * q * r \\ \frac{I_z - I_x}{I_y} * p * r \\ \frac{I_x - I_y}{I_z} * p * q \end{bmatrix}$$

Where I_x, I_y, I_z are components of the inertia vectors and p and q are the previous angular velocities.

With the physical model just obtained we can so write the functions to update the drone state in the simulation space. The drone state is described by the r_f, Ω_f, V_b, A_b vectors plus the $\dot{r_f}$ and $\dot{\Omega}_f$ vectors in order to avoid using derivatives into the code. Moreover, also the vector containing the rotational velocities of the rotors w_i is held into the state of the drone.

Define k the step index so that the current step is k+1. The first value of the state to update is the velocity in the drone body fixed frame:

$$A_b(k+1) = A_b(k) + \Delta t * \dot{A}_b(k+1)$$

Then compute the fixed frame angular velocity and update the angles in the fixed frame:

$$\dot{\Omega}(k+1) = W * A_b(k+1)$$

$$\Omega(k+1) = \Omega(k) + \Delta t * \dot{\Omega}(k+1)$$

Finally starting from the thrust T compute the new linear acceleration, speed and position:

$$\ddot{r_f}(k+1) = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = -\begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} + \frac{1}{m} * R * \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix}$$

$$\dot{r_f}(k+1) = \dot{r_f}(k) + \Delta t * \ddot{r_f}(k+1)$$

$$r_f(k+1) = r_f(k) + \Delta t * \dot{r_f}(k+1)$$

At this point the drone state is fully update.

Ball model

The ball model is much more simple respect to the one of the drone: is represented by a rigid body perfectly symmetric along all of the three axis with 3 DOF. Since for the purposes of this project there is no need to deeply investigate the ball dynamic its state is made by the position and speed in the inertia fixed frame. In the simulation the ball speed is update randomly by a dedicated task so the only thing to do at every simulation step is to update the ball position accordingly:

$$ball_{pos}(k+1) = ball_{pos}(k) + \Delta t * ball_{speed}(k+1)$$

Image recognition

Starting from the image provided by Unreal Engine there is to recognize the shape of the ball and compute it's features (diameter and position in the image) that will be used to compute the exact position of the ball in the environment respect to the drone.

To speed up the identification of the ball the algorithm exploits the last position of the ball in the image and makes a research in a reduced size window centered in that position, if the ball is not found the window takes the whole image size.

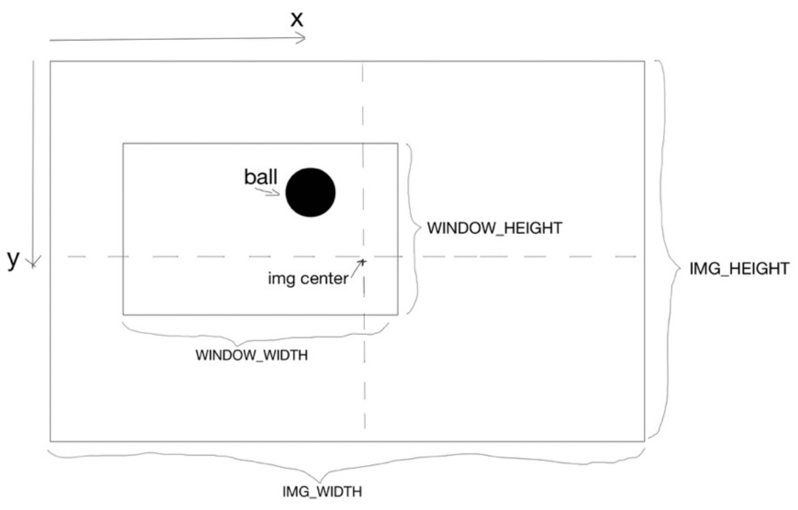


Figure 3 - Windowed ball search

To find the ball the algorithm looks for black pixels (RGB value equal to (0, 0, 0)), in particular makes the assumption that in our environment only the ball is black and that there are no reflexes on the ball (so it's color is uniformly black). This allows to simplify the algorithm making it to look to four fundamental pixels: top, bottom, most left and right. To verify that the four points represents the ball the program checks the two diameters (North-South and East-West) to verify that the difference between them is under the maximum allowed error (this is needed because when UE renders the image the ball could be not a perfect circle so a tolerance is required in order to avoid false negative); is important to set carefully this value in order to avoid false negatives and false positives, to do that I have done some experiments and the optimal value for MAX_ERR is of 10 pixels difference between the horizontal and vertical diameters.

Ball position

To compute the position of the ball respect to the drone I have exploited some simple geometry:

$$dist_{ref} : diam_{ref} = dist : diam \quad \rightarrow \quad dist = \frac{dist_{ref}}{diam_{ref}} * diam$$

Call $R = \frac{dist_{ref}}{diam_{ref}}$ the ratio between the distance from drone and the diameter in the reference image. To compute R, I made an experiment placing the ball at 300 cm of distance from the drone, from the previous step I know the diameter (39) and R is 7.692 (so I can choose to approximate that value with 7.7).

Then I have to consider the horizontal and vertical displacement respect to the drone, to do that every time the algorithm computes the ratio between pixels and centimeters using the informations about the ball diameter:

$$\frac{px}{cm} = \frac{diam_{px}}{diam_{cm}} \quad \rightarrow \quad cm = \frac{diam_{cm}}{diam_{px}} * px$$

This equation gives the number of centimeters that are represented by a given number of pixels in the plane passing through the ball center, this value changes with the distance of the ball from the drone.

Now so since we know the position of the center of the ball in the image we can compute the distance (in pixels) of the ball from the center of the image and convert that value in cm using the last equation.

This stage so we have computed the distance of the ball from the drone in all of the three axis components, all this information can now be used to control properly the drone.

Controller

Starting from the actual position of the ball respect to the drone, the first step is to compute the new desired position for the drone, fixed the parameter BALL_DST the desired distance of the drone from the ball the program computes the exact position where to place the drone.

First I needed to obtain the distance position of the ball respect to the body fixed frame: since the camera is attached to the drone, but the axes of the image recognition are flipped, is just needed to do a little adjustment:

$$\begin{cases} dist_{x_{drone}} = dist_{x_{camera}} \\ dist_{y_{drone}} = dist_{z_{camera}} \\ dist_{z_{drone}} = dist_{y_{camera}} \end{cases}$$

Now I have to move from body fixed frame to fixed frame; calling B the position of the ball, O the origin of the fixed frame, D the origin of the body frame and R the rotation matrix of the drone fixed frame respect to the fixed frame, the following relations allows to get the position of the ball in the fixed frame:

$$\overrightarrow{OB} = \overrightarrow{OD} + R * \overrightarrow{DB}$$

Where OD is known since is the position of the drone respect to the fixed frame and DB is the previously computed position of the ball respect to the drone fixed frame.

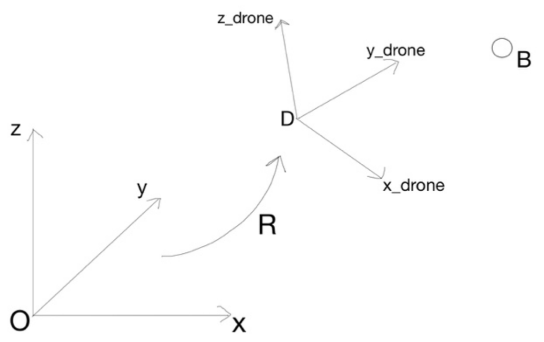


Figure 4 - Drone fixed frame respect to fixed frame

Notice: OB, OD and DB are vectors while R is a 3x3 matrix; to avoid matrix computation in the c code I computed offline the equation separating into the three components, the result is:

$$ball_x = drone_x + \cos(roll) * \cos(yaw) * dist_{x_{drone}} - (\cos(roll) * \sin(yaw) - \cos(yaw) * \sin(roll) * \sin(pitch)) * dist_{y_{drone}} + (\sin(roll) * \sin(yaw) + \cos(roll) * \cos(yaw) * \sin(pitch)) * dist\_(z\_drone\ )$$

$$ball_y = drone_y + \cos(pitch) * \sin(yaw) * dist_{x_{drone}} + (\cos(roll) * \cos(yaw) + \sin(roll) * \sin(pitch) * \sin(yaw)) * dist_{y_{drone}} - (\cos(yaw) * \sin(roll) - \sin(roll) * \sin(yaw) * \sin(pitch)) * dist_{z_{drone}}$$

$$ball_z = drone_z - \sin(pitch) * dist_{x_{drone}} + \cos(pitch) * \sin\ (roll) * dist_{y_{drone}} + \cos(roll) * \cos(pitch) * dist_{z_{drone}}$$

Now the exact position of the ball in the reference frame is known and so is possible to compute the desired position for the drone. The goal is to place the drone at the same height of the ball at a distance BALL_DST.

$$dx = ball_x - drone_x, \quad dy = ball_y - drone_y$$

$$\alpha = \tan^{-1}\left(\frac{dy}{dx}\right)$$

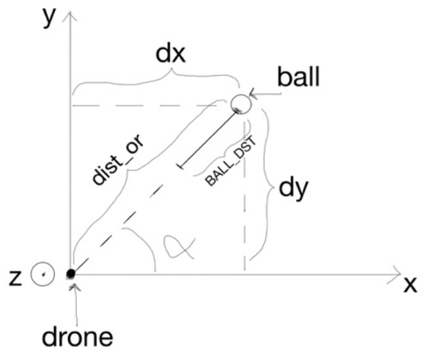


Figure 5 - Drone desired position

The actual distance on the horizontal plane of the ball from the drone is $dist_{or} = \frac{dx}{\cos\alpha}$, so using some trigonometry is possible to compute the new position of the drone:

$$\begin{cases} des_x = drone_x + (dist_{or} - BALL\_DST) * \cos(\alpha) \\ des_y = drone_y + (dist_{or} - BALL\_DST) * \sin(\alpha) \\ des_z = ball_z \end{cases}$$

Since the drone should be horizontal looking to the ball, the desired orientation of the drone is:

$$\begin{cases} roll = 0 \\ pitch = 0 \\ yaw = \alpha \end{cases}$$

That desired position and orientation is given as input to digital controller composed by two loops: the external one computes the thrust and the torques to command in order to achieve the target position, while the internal one computes the required forces of each to do that. Both loops are done using PD controllers which gains have been found using MATLAB simulation and then adjusted with some experiments. In order to realize a realistic simulation each rotor cannot exceed a maximum speed and consequently a maximum force.

The output forces and torques of the controller are then converted into angular speeds for the four rotors of the drone, using the inverse of the Force Allocation Matrix (M) seen in the drone model, in order to give these as input to the vehicle.

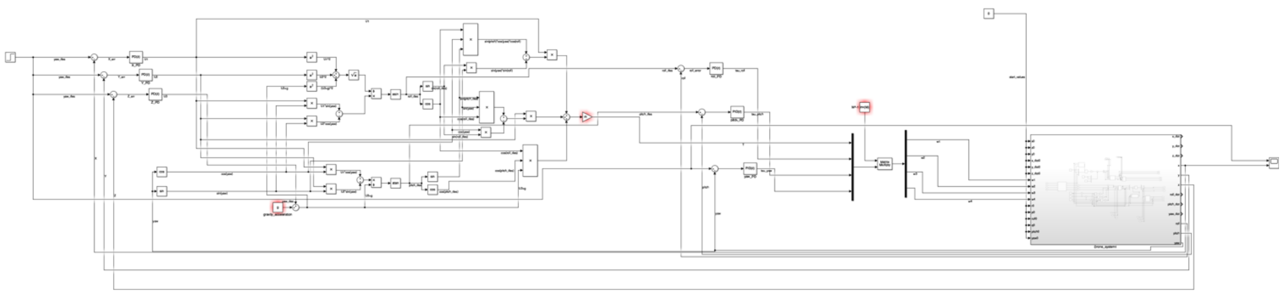


Figure 6 - controller scheme (MATLAB)

Design choices

World

To simplify the render operation the world where to simulate is flat and without obstacles (excluded the ground), moreover in order to simplify the ball recognition algorithm the world is of a color very different from the ball and the shades are very light.

Ball

The color of the ball is set to be fully black without reflexes or shades on it in order to simplify the recognition algorithm. Another important aspect is the ball maximum speed which is limited from -MAX_BALL_SPEED to MAX_BALL_SPEED; this is due to simplify the controller.

User interfaces

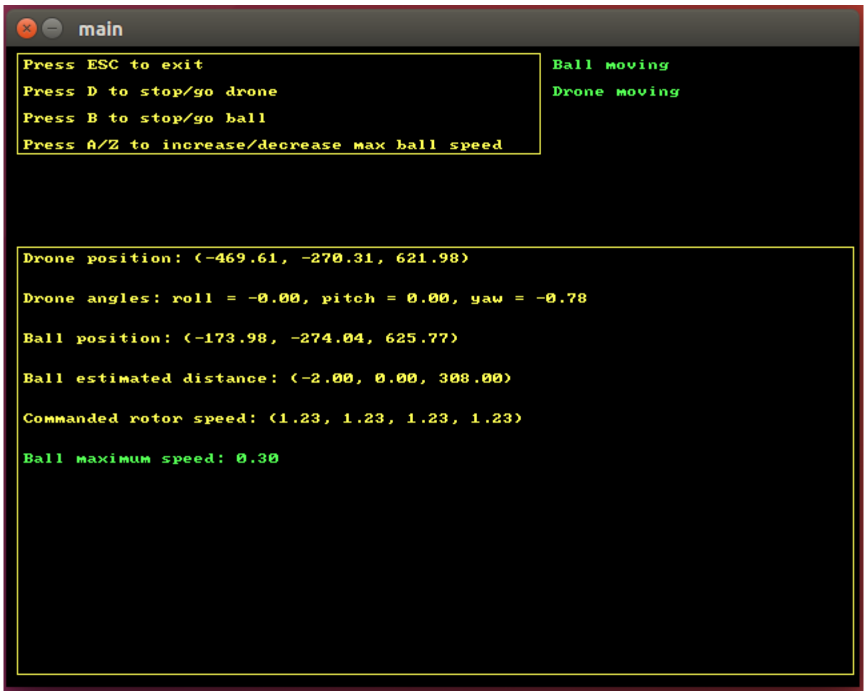
There are two interfaces for the user: the 3D world simulator and the control panel. The 3D simulator is realized with using Unreal Engine 4.19, is an application that runs on MacOS, Linux or Windows. The control panel is a simple interface that is realized using Allegro Library v.3 and runs on Linux. When the simulator is working both user interfaces are available for the user, to do that the best solution is to use two different OS (eventually emulated): one running Linux and one running MacOS, Linux or Windows. There are no special hardware prerequisites in order to run the control panel, while in order to run the 3D simulator there are some recommended hardware and software requirements that are needed by the engine to work correctly; these requirements are listed below:

Platform	OS Version	Processor	Memory	Video Card
MacOs	High Sierra 10.3.5	Quad-core Intel 2.5 GHz or faster	8 GB RAM	Metal 1.2 Compatible Graphics Card
Linux	Ubuntu 18.04	Quad-core Intel or AMD 2.5 GHz or faster	32 GB RAM	NVIDIA GeForce 960 GTX or higher with latest NVIDIA binary drivers
Windows	Windows 10 – 64bit	Quad-core Intel or AMD 2.5 GHz or faster	8 GB RAM	DirectX 11 Compatible

Table 1 - Recommended requirements (UE 4 documentation)

User panel

This panel allows the user to control the status of the simulation, change ball speed, exit and start or stop the movement of the ball and drone. The interface is intuitive: on the top left displays the commands to control the program. On the top right is shown the status of the ball and of the drone that can be "stopped" or "moving" depending on the user inputs. Between the two boxes can appear eventual error messages about images receiving. On the bottom box are displayed some useful data about the ongoing simulation such as drone and ball position, the estimated distance of the ball from the drone (respect to drone fixed frame), the rotors speeds commanded by the controller and the actual maximum ball speed selected by the user. An image of the control panel is shown below:



```
Press ESC to exit                              Ball moving
Press D to stop/go drone                       Drone moving
Press B to stop/go ball
Press A/Z to increase/decrease max ball speed

Drone position: (-469.61, -270.31, 621.98)

Drone angles: roll = -0.00, pitch = 0.00, yaw = -0.78

Ball position: (-173.98, -274.04, 625.77)

Ball estimated distance: (-2.00, 0.00, 308.00)

Commanded rotor speed: (1.23, 1.23, 1.23, 1.23)

Ball maximum speed: 0.30
```

Figure 7 - Control panel

3D simulator interface

This interface can be used to follow the evolution of the simulation in the 3D world. The user here sees the drone camera perspective and the only interaction is the quit of the perspective that can be performed by pressing ESC.

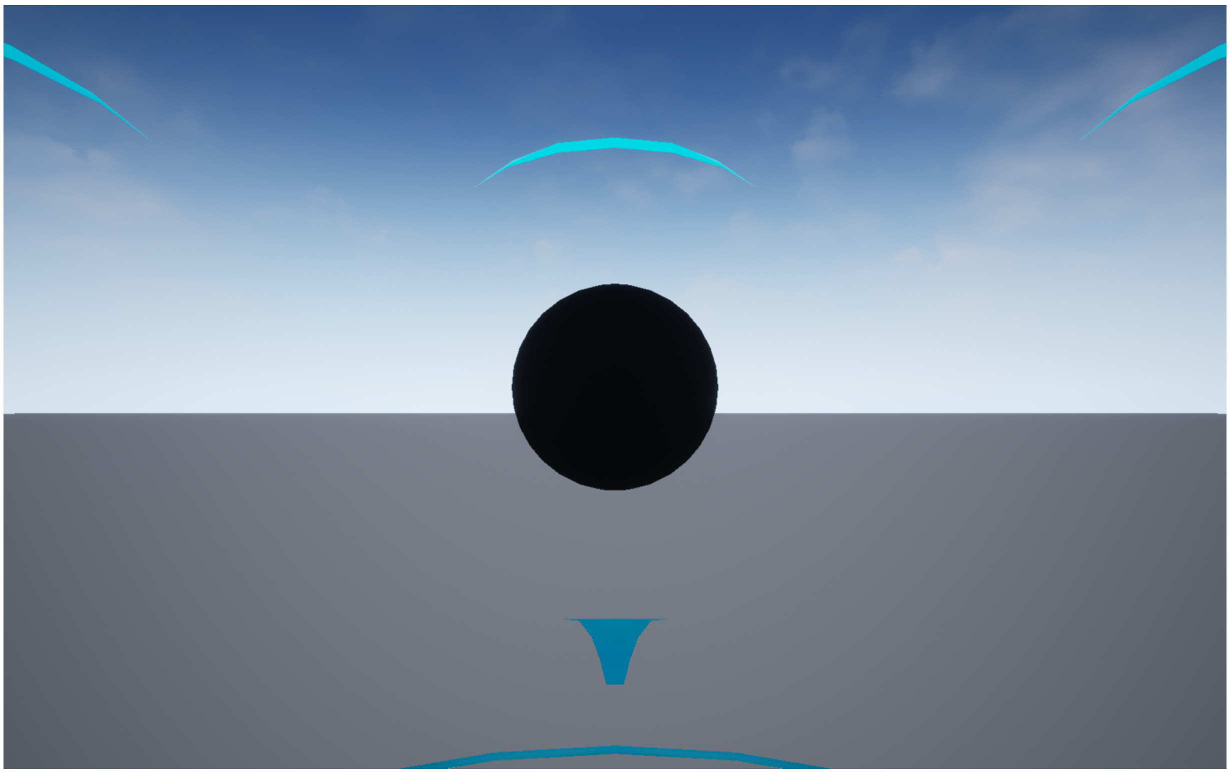


Figure 8 - 3D Simulator interface

Tasks and data structures

To simulate the whole environment are involved some concurrent activities to compute the physical evolution of the system, his control and manage the network transmissions. All of this works are done by eight different tasks running concurrently:

- UDP image packets receiving (RECV)
- Drone physic evolution simulation (DRONE)
- Drone control computation (CONTROLLER)
- Ball physic evolution simulation (BALL)
- Ball recognition in the image (RECOGNITION)
- UDP position packets sending (SEND)
- Ball speed random update (INPUT)
- User interaction (GUI)

UDP image packets receiving

This task is in charge of receive the image packets sent by the 3D world simulator. The task waits for a header packet and then collects all of the following packets until completes to receive the image bitmap, then saves that into a shared variable and signals that a new image is available, then restarts the process.

Drone physic evolution simulation

This thread is responsible to compute the evolution of the position, velocity and acceleration of the drone in the environment following the physical laws described into the drone model.

Drone control computation

The drone control computation task will simulate the behavior of a microcontroller responsible to give to the motor the correct instructions to make the drone move in the desired position. This task takes as input the position of the ball respect to the drone and knows the position of the drone in the fixed frame, starting from that data computes the position of the ball in the fixed frame, then the target position where to place the drone and finally the required rotational speed to require to each rotor to move in that position. Also stability of the drone is guaranteed by this task.

Ball physic evolution simulation

As for the drone the evolution in time of the position of the ball in the environment is done by this thread following the physical laws described into the ball model.

Ball recognition in the image

This thread takes the image received by the RECV task and applies the recognition algorithm to find if there is a ball in the image and if so his position respect to the drone. The output is placed into a shared variable.

UDP position packets sending

To provide a fluid representation of the simulation in the 3D world simulator this task sends an UDP packet containing the actual position of the ball and of the drone to the simulator. To have a smooth visualization of the simulation the human eye needs at least 24 frames for second, so this task sends 30 position update packets for seconds this way is possible to have a fluid visualization even if some packet gets lost.

Ball speed random update

This task randomly updates the speed of the ball respect to each axis; as previously mentioned in the ball model the speed of the ball along each axis is randomized between -MAX_BALL_SPEED and MAX_BALL_SPEED.

User interaction

The user interaction task is in charge to update the control panel user interface with the latest information about the program, moreover checks for user commands to modify simulation parameters or terminate it by setting the end flag and signaling it to main with a dedicated semaphore.

Data structures

To realize a correct simulation the threads composing the program needs to exchange information among them; to do this are needed some data structures:

- drone_state – this struct contains all of the information about the state of the drone: position, speed and acceleration in both of the frames plus the rotational speed of each rotor;
- ball_state – this struct contains the info about position and speed of the ball;
- image_info – contains the information about the image received (size, width and height) and the vector containing the bitmap received.

Over to these structures, thread shares also other simple global variables all protected by mutexes.

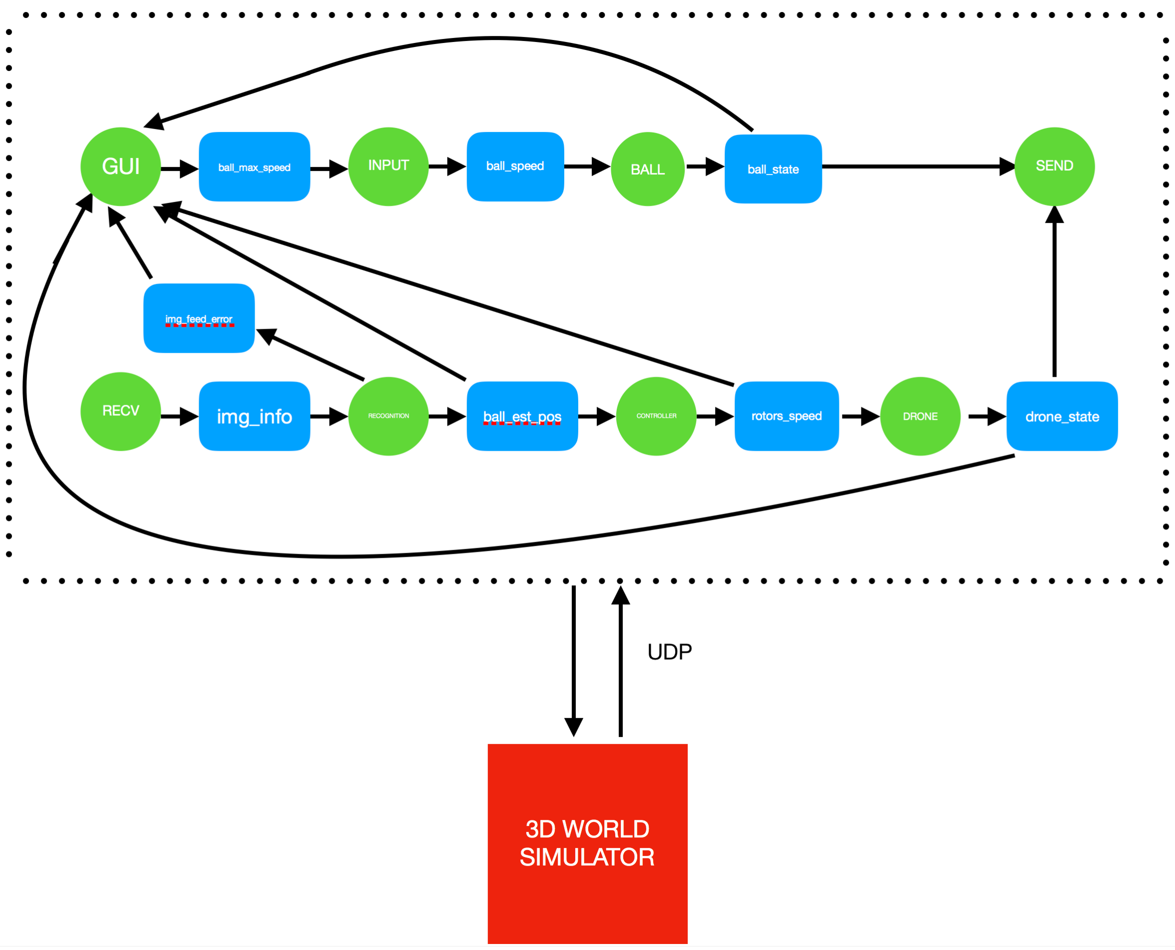


Figure 9 - Task interaction graph

Timing and schedule

In this program eight different threads executes at the same time; in reality this is not really what happens since it would require having 8 different dedicated CPUs, so is required to have a scheduling algorithm that shares the available CPU/CPUs among all of the involved threads. In the environment used to test the application the Linux part runs into a virtual machine with just one processor that so have to be shared among the eight involved threads.

As previously mentioned to have a smooth visualization of a video is needed to have at least 24 updates for second, so to be resistant against the loose of some packets is needed to send 30 packets for second. This means that the SEND task period should be $\frac{1\ s}{30\ packet} = 0.033\ s = 33\ ms$

Since it would be useless to send a packet containing the position update would be useless without any new data to display also the DRONE and BALL tasks should have the same period. These tasks are also some of the most important and so the ones with the second highest priority.

For simplicity consider that the slowest component involved in the control is the GPS that acquires the position of the drone with a frequency of 50 Hz, so the controller should have a period of 20ms, the same reasoning is valid also for the image recognition which is a component connected to the controller, these elements are of secondary importance respect to the previous ones so have a lower priority.

The period of the INPUT task, since in useless to update too fast the speed of the ball, can be realistically of 40ms, moreover this task is not much important for the system so have the lowest priority.

The RECV task is event driven tasks since is activated by the arrival of a packet, about its priority the RECV task is the most important in the system since is vital for the correct tracking of the ball and so have the highest priority (5).

Since is not needed to be very fast in user interaction the GUI task can check for user input and update the control panel graphic every 40ms, moreover since this task is not important for the correct

With all of these considerations is obviously needed a real-time scheduling algorithm to ensure the respect of deadlines (that for simplicity are chosen equals to the periods). Looking to the scheduling algorithms available in the Linux kernel the best choice is SCHED_FIFO since allows the threads with the same priority to be executed following the FIFO algorithm and each task can do his execution unless another task with higher priority preempts it.

Testing environment

The whole system has been tested on a MacBook Pro 15" of 2017 with an Intel Core i7 3,1 GHz processor, 16 GB of RAM and a graphic card Radeon Pro 560 of 4GB running MacOS High Sierra 10.13. The control panel runs into a virtual machine created with Parallels Desktop inside the host system, while the 3D world simulator runs into the host system. The VM have one dedicated processor and 1 GB of RAM running Ubuntu 16.04. Communication among the two machines is handled by Parallels Desktop that creates a shared network between them.

Results and future work

Ball recognition algorithm

In this work the ball recognition algorithm developed is very simple and is based on some assumptions: the ball color is uniform and without reflexes and shades on it and moreover that color is not present in the surrounding environment. In a real context these constraints are too strict, so it would be needed to develop a more accurate algorithm capable of detecting the target into a noisier context. Moreover, the target to follow could become no more a ball, but for example a person or a car; this could be useful in a wide number of real application where it could be useful to have an autonomous vehicle capable of follow a given object (for example for surveillance or to record the actions of an athlete).

Drone controller

The employed controller has a slow response to the ball movement, this is due to the complexity in tuning the two levels of PD controllers to achieve the maximum performances without losing stability of the vehicle. This slow response required to reduce the speed of the ball in order to allow the drone to follow it. This limit in a real context would be critical and would make the system useless, so it would be strategic to improve the performances of the controller in order to achieve a faster response without losing stability. This could be achieved by synthetizing a specified controller or more realistically by moving from PD to PID controllers.

UDP transmission

Actually, the image sent from the 3D world simulator is in RAW format, this simplifies the analysis of the image to detect the ball, but respect to a compressed format requires much more packets to deliver the image through the network. Moreover, the implemented protocol discards the whole image if another header packet is received before the image; in the tested environment where the loss of packets could be considered negligible this is not a problem, but in a real context this would become critical since the system could receive rarely a full image making so nearly impossible for the controller to follow correctly the ball. For this reason future works will include changing the format of the image from RAW to a compressed one (for example JPEG) in order to reduce the number of packets to send through the network and the improvement of the transmission algorithm in order to avoid to discard the whole image, but simply to require to the simulator to send another time the missing packet (this could be done by moving from UDP to TCP renouncing to the more speed guaranteed by the UDP solution).

Bibliografia

I. W. R. Beard, "Quadrotor dynamics and control." Brigham Young University, 2008.
II. D. Casini, "Quadrotor control with smartphone" MECS, 2015