

# **"Visual control of the Parrot drone with OpenCV, Ros and Gazebo Simulator."**

**12.06.2016**

**Artur Banach**

## Table of contents

1. <i>Introduction</i> .....	3
2. <i>ROS</i> .....	4
3. <i>Parrot</i> .....	5
4. <i>Gazebo simulator</i> .....	6
5. <i>OpenCV and drone movement - code analysis</i> .....	10
6. <i>Results of work</i> .....	21
7. <i>Summary</i> .....	22
8. <i>Bibliography</i> .....	23

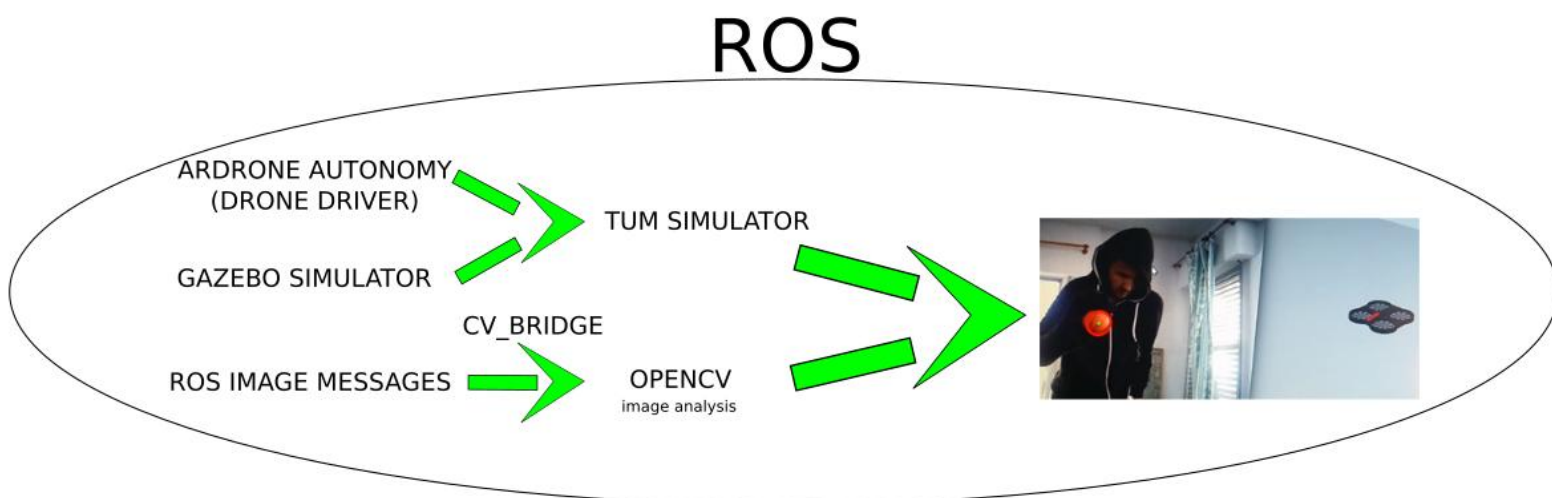
## 1. Introduction

Drones are becoming more and more popular these days. There are multiple tasks and ideas that can be implemented to them. One of them is the idea described in that work.

The aim of the project was creating a software in C++ in ROS that will control the Parrot Drone simulated in Gazebo Simulator. The drone movements are stimulated by the orange ball movements in front of the camera (the picture below). As it is presented on the diagram below, there were needed: Ardrone Autonomy Package (drone driver), Gazebo Simulator, Tum Simulator (connection of two previous ones), and cv\_bridge in OpenCV library, which converts ros image messages to OpenCV objects. Then the objects can be modified in OpenCV library.

The evaluation of project had two main phases. First of them was to configure Ardrone Autonomy driver and Gazebo Simulator. Second step of that phase was to connect two previous ones by configuring TUM Simulator. Second main phase was to write a program that converts ros image messages from the camera into openCV images, analyzes them and depending on the movement of the ball controls the drone movement.

The target was achieved and the drone is being controlled by the ball in front of the camera.



*Picture 1 - general idea*

## 2. ROS

Robot Operating System (ROS) is a collection of software frameworks for robot software development. ROS provides standard operating system services such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management.

First step in the project was to install ROS on Linux OS. [1] The installed version was ROS Indigo. Installation tutorial is provided on the website: <http://wiki.ros.org/indigo/Installation/Ubuntu>.

After installing ROS software with the command:

```
sudo apt-get install ros-indigo-desktop-full
```

it was necessary to create a ROS Workspace, which as the name implies, is the area of computer memory where all the files connected to ROS projects are stored. It was done with commands:

```
mkdir -p ~/catkin_ws/src  
cd ~/catkin_ws/src  
catkin_init_workspace
```

Then it was possible to build workspace. It is important because later after every change in your program we need to use that command to compile it:

```
cd ~/catkin_ws/  
catkin_make
```

When the workspace is created, it is needed to create Catkin Package. This is a package where all the files connected to the one specific project are stored. The recommended method of working with catkin packages is creating it inside of catkin workspace. For a package to be considered a catkin package it must meet a few requirements:

- The package must contain a catkin compliant package.xml file.
- The package must contain a CMakeLists.txt which uses catkin.
- There can be no more than one package in each folder.

To create Catkin Package it was needed to write:

```
cd ~/catkin_ws/src  
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

All the dependencies for the package are stored in package.xml. In the picture below, we can see the code from that file with all needed dependencies.

```
<!-- The *_depend tags are used to specify dependencies -->
<!-- Dependencies can be catkin packages or system dependencies -->
<!-- Examples: -->
<!-- Use build_depend for packages you need at compile time: -->
<!--   <build_depend>message_generation</build_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!--   <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use run_depend for packages you need at runtime: -->
<!--   <run_depend>message_runtime</run_depend> -->
<!-- Use test_depend for packages you need only for testing: -->
<!--   <test_depend>gtest</test_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>ardrone_autonomy</build_depend>
<build_depend>geometry_msgs</build_depend>
<build_depend>roscpp</build_depend>
<build_depend>sensor_msgs</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>geographic_msgs</build_depend>
<build_depend>image_transport</build_depend>
<build_depend>cv_bridge</build_depend>
<run_depend>ardrone_autonomy</run_depend>
<run_depend>geometry_msgs</run_depend>
<run_depend>roscpp</run_depend>
<run_depend>sensor_msgs</run_depend>
<run_depend>std_msgs</run_depend>
<run_depend>geographic_msgs</run_depend>
<run_depend>image_transport</run_depend>
<run_depend>cv_bridge</run_depend>
```

It is very important to understand basic parts of ROS as node, topic and message. Node is an executable file within ROS package. ROS nodes use ROS client library to communicate with other nodes. Nodes can publish or subscribe a topic, while messages are data types used when subscribing or publishing the topic.

After a detailed tutorial introduction to ROS and obtaining basic knowledge how to use it, it was possible to install Ardrone Autonomy package, Gazebo Simulator and finally TUM simulator. Details are explained in next chapters.

### 3. Parrot

Parrot AR.Drone is a remote controlled flying quadcopter built by the French company Parrot. The drone is designed to be controlled by mobile or tablet operating systems such as the supported iOS or Android within their respective apps or the unofficial software available for Windows Phone, Samsung BADA and Symbian devices. In the picture below there is Parrot AR.Drone presented.



*Picture 2 - Parrot Ardrone*

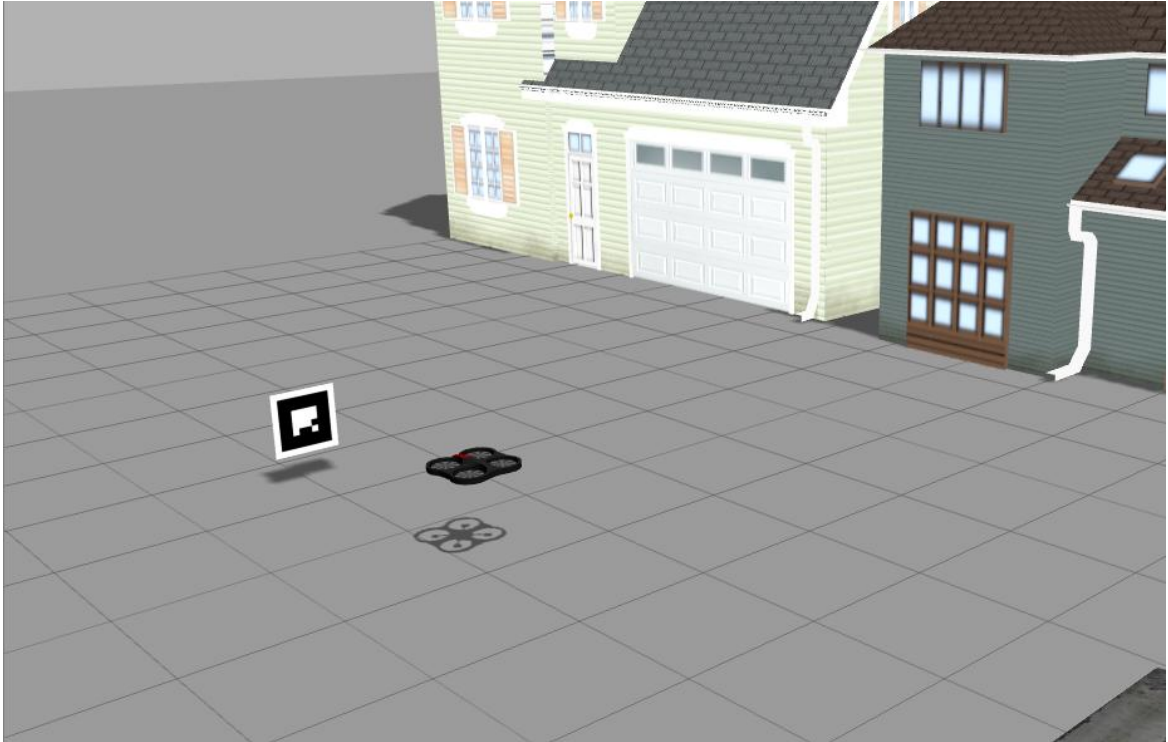
The heart of the drone is ARM Cortex A8 1 GHz 32-bit processor and RAM DDR2 1 GB at 200 MHz. Operation system is Linux 2.6.32. The drone has USB 2.0 interface, gyroscope with accuracy of  $2,000^{\circ}$ /second, accelerometer with accuracy of  $\pm 50$  mg, magnetometer with accuracy of  $6^{\circ}$ , pressure sensor with accuracy of  $\pm 10$  Pa, altitude ultrasound sensor and vertical camera QVGA 60 FPS to measure the ground speed. There is also embeded HD Camera 720p 30fps. When it comes to propulsion there ar 4 brush-free DC motors with 14.5 watts and 28,500 rev/min. The weight if the whole drone is around 420 g. The drone is usually equipped with 1 1000 mAh battery. There is also AR.FreeFlight mobile aplication provided that saves photos, navigation data and videos from the drone.

To simulate that specific drone it was necessary to install ROS Ardrone Autonomy package, which is a driver for Parrot drone. To install that package it is needet to use commands:

```
cd ~/catkin_ws/src
git clone https://github.com/AutonomyLab/ardrone_autonomy.git -b indigo-devel
cd ~/catkin_ws
rosdep install --from-paths src -i
catkin_make
```

#### **4. Gazebo simulator**

Next step was to install Gazebo Simulator, which is a 3D dynamic simulator with the ability to simulate various robots in indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces. In the picture below there is presented how it looks like while simulating the Parrot drone in Gazebo Simulator.



*Picture 3 - Gazebo simulator with Parrot Ardrone*

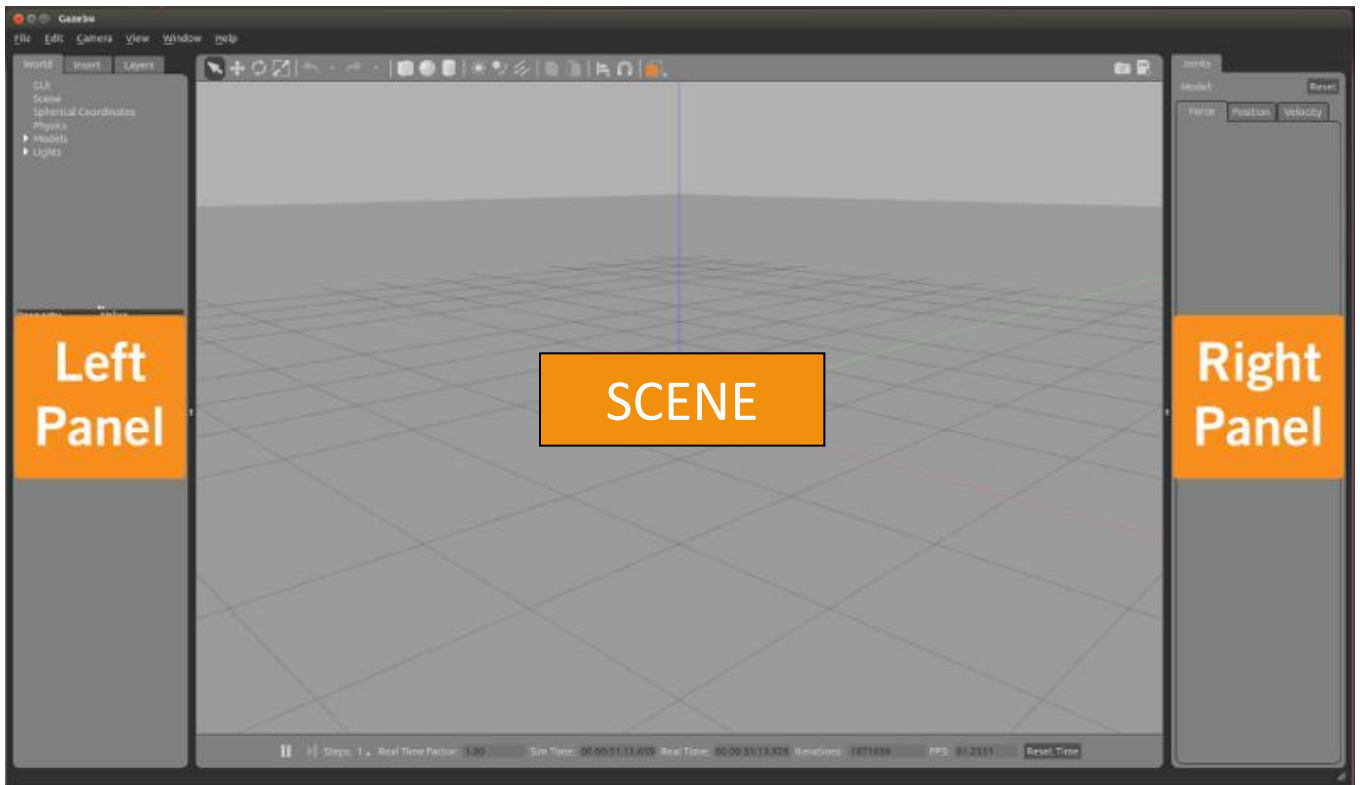
There are specific system requirements:

- Linux OS (Ubuntu Trusty or later)
- A dedicated GPU (Nvidia cards work well)
- A CPU that is at least Intel i5, or equivalent
- 0.5 GB of free disk space

Installation of Gazebo Simulator was done with the command [3]:

```
sudo apt-get install ros-indigo-gazebo-ros-pkgs ros-indigo-gazebo-ros-control
```

Gazebo interface consists of multiple sections. First of them - scene is the main part of the simulator. This is where the simulated objects are animated and interact with the environment. Second section are panels.



Picture 4 - Gazebo interface

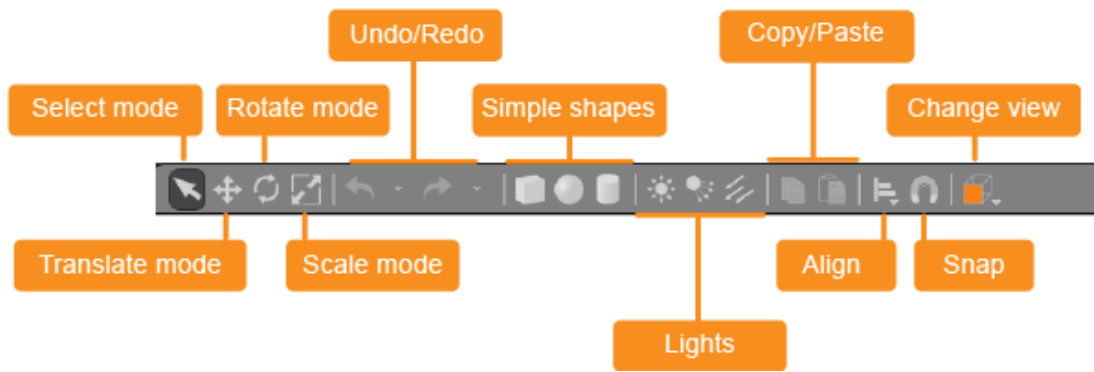
The left panel appears by default when you launch Gazebo. There are three tabs in the panel:

- **WORLD:** displays the models that are currently in the scene, and allows you to view and modify model parameters, like their pose
- **INSERT:** The Insert tab is where you add new objects (models) to the simulation.
- **LAYERS:** The Layers tab organizes and displays the different visualization groups that are available in the simulation.

The right panel is hidden by default. It is used to interact with the mobile parts of a selected model (the joints). If there are no models selected in the Scene, the panel does not display any information.

Third section are two toolbars. One of them is located above the Scene and one below. The upper toolbar is a main one and includes most-used options: select, move, rotate, scale, create shape, copy and paste:





Picture 5 - Gazebo Toolbox

The Bottom Toolbar is useful during the simulation. It displays simulation time, real time and Real Time Factor, which is a relationship between two previous ones. The state of the world in Gazebo is calculated once per iteration. You can see the number of iterations on the right side of the bottom toolbar. Each iteration advances simulation by a fixed number of seconds, called the step size. [4]

This is not the end of installing software. What is needed more is to connect the Parrot driver and Gazebo Simulator. It can be achieved by installing TUM Simulator, which is an implementation of Gazebo Simulator and Ardrone Autonomy package. Installing that package allows to run the drone in an artificial world in Gazebo Simulator (as shown in the picture above). TUM Simulator installation was made with commands[5]:

```

# cd into ros root dir
roscd

# clone repository
git clone https://github.com/tum-vision/tum_simulator.git

# add to ros path (if required)
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:`pwd`/tum_simulator

# build package
rosmake cvg_sim_gazebo_plugins
rosmake message_to_tf
  
```

Now when there was TUM Simulator installed, it was possible to run the drone in the Gazebo Simulator. It was done with the following command:

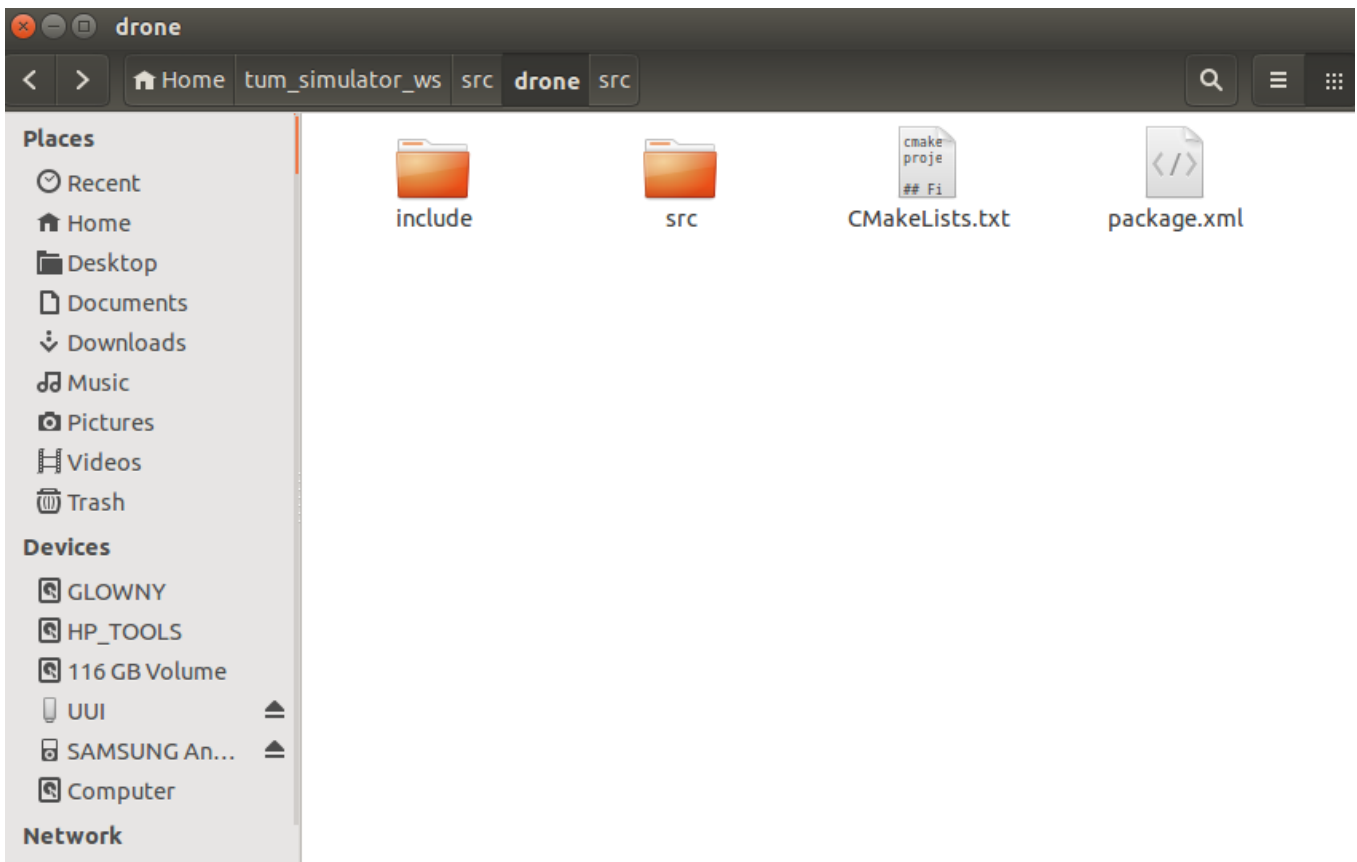
```

roslaunch cvg_sim_test 3boxes_room.launch
  
```

To see how it moves it was necessary to develop the program, which is described in the next chapter.

## 5. OpenCV and drone movement - code analysis

First of all it is important to explain how the ROS file organization works. In this project the package was called "drone". Inside of the package there is a folder "src", where you can create main file that will be executed during the compilation.



Picture 6 - Ros package

In the file CMakeList.txt it should be stated which cpp files should be executed. In that project the executed file is "main.cpp". In the picture below there is code from CMakeList.txt presented. First there are written names of packages that need to be found. Then are declared c++ file that are going to be compiled with every "catkin\_make". The main.cpp is given with name "main" (useful when running the program). The "key" program was just a trial program. Then specific libraries are

linked to executable program. In our case they are standard "catkin\_LIBRARIES" and "OPENCV\_LIBRARIES".

```
cmake_minimum_required(VERSION 2.8.3)
project(drone)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  ardrone_autonomy
  geometry_msgs
  roscpp
  sensor_msgs
  std_msgs
  geographic_msgs
  image_transport
  cv_bridge
)

#####
## Build ##
#####

## Specify additional locations of header files
## Your package locations should be listed before other locations
# include_directories(include)
include_directories(
  ${catkin_INCLUDE_DIRS}
)

include_directories(SYSTEM ${OpenCV_INCLUDE_DIRS})

## Declare a C++ executable
add_executable(main src/main.cpp)
add_executable(key src/key_control.cpp)
## Add cmake target dependencies of the executable
## same as for the library above
# add_dependencies(drone_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

## Specify libraries to link a library or executable target against
target_link_libraries(key
  ${catkin_LIBRARIES}
)

target_link_libraries(main
  ${catkin_LIBRARIES}
)

target_link_libraries(main
  ${OPENCV_LIBRARIES}
)
```

*Picture 7 - CMakeList.txt*

Now it is time to analyze the code from main.cpp file. The code consists of three parts.

First one is the drone-movement part. There are stated all the function that publish messages to the simulation node to move the drone. As it is presented on the printscreens below, every function is publishing another topic with different message (different velocity in different direction). It stimulates in which direction and how fast the drone will move.

In the beginning there are defined topics. 4 of them are to publish and one is a service. Their functions are explained later.

Then the first function is being defined. It is a function of class `geometry_msgs::Twist`, and the message `msg_vel` (inside) is of the same type, which means it is going to change velocities of the drone movement.

The next function calls `flattrim` service that allows the drone to fly with stability on one height.

Function `takeoff()` is responsible for starting the drone. There is created a message "empty" of type `std_msgs::Empty`. Then again a message `msg_vel` is being created. Topic `topicstakeoff` is being published with an empty message (this is how to start the drone). Then the topic with geometry message is being published and it stops the drone in the air.

```
//-----DRONE FUNCTIONS-----

ros::Publisher topicstakeoff;
ros::Publisher topicclanding;
ros::ServiceClient serviceflattrim;
ros::Publisher cmd_vel;
ros::Publisher topicinfo;
float longitud, latitud, altitud;

geometry_msgs::Twist changeTwist(float x, float y, float z, float turn){
    geometry_msgs::Twist msg_vel;
    msg_vel.angular.x = 0;
    msg_vel.angular.y = 0;
    msg_vel.angular.z = turn;
    msg_vel.linear.x = x;
    msg_vel.linear.y = y;
    msg_vel.linear.z = z;
    return(msg_vel);
}

void ajuste (void){
    // Ajuste horizontal (Calibracion)
    // Llamada al servicio que permite el ajuste (Sincrono)
    std_srvs::Empty srvflattrim;
    serviceflattrim.call(srvflattrim);
}

void takeoff (void){
    // Despegue y lo dejamos haciendo hovering
    // Publicamos en el topic takeoff (Asincrono)
    std_msgs::Empty empty;
    geometry_msgs::Twist msg_vel;
    topicstakeoff.publish(empty);
    printf("Starting...\n");
    usleep(250000);
    printf("Cambiando velocidad para hovering...\n");
    msg_vel = changeTwist(0,0,0,0);
    cmd_vel.publish(msg_vel);
}
```

*Picture 8 - main.cpp, drone movement functions*

In the function `land()` just the topic `topicclanding` (defined later in main function) with an empty message is being published. That is enough to land the drone.

Now we are getting to moving the drone in the air. First function responsible for the movement forward is called `forwardx()` and as every next function has 3 steps. First one is creating a message `msg_vel` (as the name implies it contains information about velocities). Second step is changing velocity with the function `changeTwist()`. When number

appears as a first argument of that function it means drone will move in x direction, if as a second argument -> y direction, if as third argument -> z direction and if the fourth then the drone will rotate. So in the function forwardx() 1 appears as a first argument. Then the topic cmd\_vel is being published with the msg\_vel message. Analogically things happen in every following function. When the direction needs to be reversed then the argument is simply negative.

```

void land (void){
    // Aterrizar
    // Publicamos en el topic land
    std_msgs::Empty empty;
    topiclanding.publish(empty);
}
void forwardx (void){
    geometry_msgs::Twist msg_vel; //info velocity
    msg_vel = changeTwist(1,0,0,0);
    cmd_vel.publish(msg_vel);
}
void forwardy (void){
    geometry_msgs::Twist msg_vel; //info velocity
    msg_vel = changeTwist(0,1,0,0);
    cmd_vel.publish(msg_vel);
}
void backx (void){
    geometry_msgs::Twist msg_vel; //info velocity
    msg_vel = changeTwist(-1,0,0,0);
    cmd_vel.publish(msg_vel);
}
void backy (void){
    geometry_msgs::Twist msg_vel; //info velocity
    msg_vel = changeTwist(0,-1,0,0);
    cmd_vel.publish(msg_vel);
}
void up (void){
    geometry_msgs::Twist msg_vel; //info velocity
    msg_vel = changeTwist(0,0,1,0);
    cmd_vel.publish(msg_vel);
}
void down (void){
    geometry_msgs::Twist msg_vel; //info velocity
    msg_vel = changeTwist(0,0,-1,0);
    cmd_vel.publish(msg_vel);
}
void stop(void){
    geometry_msgs::Twist msg_vel; //info velocity
    msg_vel = changeTwist(0,0,0,0);
    cmd_vel.publish(msg_vel);
    //printf("stopping...\n");
}

```

*Picture 9 - main.cpp, drone movement functions*

Second part of the program is the image analysis and drone control. Here the most important things happen. The whole image analysis is based on cv\_bridge that converts ROS images into OpenCV images. Details how to use cv\_bridge are available on website:[http://wiki.ros.org/cv\\_bridge/Tutorials/UsingCvBridgeToConvertBetween](http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetween)

ROSIImagesAndOpenCVImages. After declaring all necessary class objects and defining constructor, the callback function of the camera begins.

The image operations begin with HSV transform. It is done with OpenCV `cvtColor ()` function, which converts an image from one color space to another.

**`void cvtColor(InputArray src, OutputArray dst, int code, int dstCn=0 )`**

**Parameters:**

- **src** – input image: 8-bit unsigned, 16-bit unsigned, or single-precision floating-point.
- **dst** – output image of the same size and depth as src.
- **code** – color space conversion code (see the description below).
- **dstCn** – number of channels in the destination image; if the parameter is 0, the number of the channels is derived automatically from src and code. [6]

Hue, Saturation, Value or HSV is a color model (used in that project) that describes colors (hue or tint) in terms of their shade (saturation or amount of gray) and their brightness (value or luminance). The HSV color wheel may be depicted as a cone or cylinder.

- Hue is expressed as a number from 0 to 360 degrees representing hues of red (starts at 0), yellow (starts at 60), green (starts at 120), cyan (starts at 180), blue (starts at 240), and magenta (starts at 300).
- Saturation is the amount of gray (0% to 100%) in the color.
- Value (or Brightness) works in conjunction with saturation and describes the brightness or intensity of the color from 0% to 100%. [7]

Then the mask is created with `cvCreateMat()` function, which creates a matrix header and allocates the data:

**`cvCreateMat(int rows, int cols, int type)`**

**Parameters:**

- **rows** – Number of rows in the matrix
- **cols** – Number of columns in the matrix
- **type** – The type of the matrix elements in the form `CV_<bit depth><S|U|F><number of channels>`, where S=signed, U=unsigned, F=float. For example, `CV_8UC1` means the elements are 8-bit unsigned and there is 1 channel, and `CV_32SC2` means the elements are 32-

bit signed and there are 2 channels. [8]

Then `inRange` function, which checks if array elements lie between the elements of two other arrays is executed. Saying it more simply, that function saves to the mask only the objects with the colors that are in range of the parameters given inside of the function:

**void inRange(InputArray src, InputArray lowerb, InputArray upperb, OutputArray dst)**

- Parameters:**
- **src** – first input array.
  - **lowerb** – inclusive lower boundary array or a scalar.
  - **upperb** – inclusive upper boundary array or a scalar.
  - **dst** – output array of the same size as `src` and `CV_8U` type.

Next step is morphological closing and opening that make the detected image more uniform. They consist of dilation and erosion, the most basic morphological operations, that are used to:

- Remove noise
- Isolate individual elements and join disparate elements in an image.
- Find intensity bumps or holes in an image

Dilation consists of convoluting an image `A` with some kernel `B`, which can have any shape or size, usually a square or circle. The kernel `B` has a defined anchor point, usually being the center of the kernel. As the kernel `B` is scanned over the image, we compute the maximal pixel value overlapped by `B` and replace the image pixel in the anchor point position with that maximal value. As you can deduce, this maximizing operation causes bright regions within an image to “grow” (therefore the name dilation).[9]

**void dilate(InputArray src, OutputArray dst, InputArray kernel, Point anchor=Point(-1,-1), int iterations=1, int borderType=BORDER\_CONSTANT, const Scalar& borderValue=morphologyDefaultBorderValue() )**

- Parameters:**
- **src** – input image; the number of channels can be arbitrary, but the depth should be one of `CV_8U`, `CV_16U`, `CV_16S`, `CV_32F` or `CV_64F`.
  - **dst** – output image of the same size and type as `src`.

- **element** – structuring element used for dilation; if `element=Mat()` , a 3 x 3 rectangular structuring element is used.
- **anchor** – position of the anchor within the element; default value (-1, -1) means that the anchor is at the element center.
- **iterations** – number of times dilation is applied.
- **borderType** – pixel extrapolation method (see `borderInterpolate()` for details).
- **borderValue** – border value in case of a constant border (see `createMorphologyFilter()` for details).

Erosion is the sister of dilation. What this does is to compute a local minimum over the area of the kernel. As the kernel B is scanned over the image, we compute the minimal pixel value overlapped by B and replace the image pixel under the anchor point with that minimal value.

```
void erode(InputArray src, OutputArray dst, InputArray kernel, Point anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const Scalar& borderValue=morphologyDefaultBorderValue() )
```

**Parameters:**

- **src** – input image; the number of channels can be arbitrary, but the depth should be one of CV\_8U, CV\_16U, CV\_16S, CV\_32F or CV\_64F.
- **dst** – output image of the same size and type as `src`.
- **element** – structuring element used for erosion; if `element=Mat()` , a 3 x 3 rectangular structuring element is used.
- **anchor** – position of the anchor within the element; default value (-1, -1) means that the anchor is at the element center.
- **iterations** – number of times erosion is applied.
- **borderType** – pixel extrapolation method (see `borderInterpolate()` for details).
- **borderValue** – border value in case of a constant border (see `createMorphologyFilter()` for details). [10]

Next the gaussian blur is being put (to avoid wrong circles detection). Gaussian blurring is highly effective in removing gaussian noise from the image:

```
void GaussianBlur(InputArray src, OutputArray dst, Size ksize, double sigmaX, double sigmaY=0, int borderType=BORDER_DEFAULT )
```



- **src** – input image; the image can have any number of channels, which are processed independently, but the depth should be CV\_8U, CV\_16U, CV\_16S, CV\_32F or CV\_64F.
  - **dst** – output image of the same size and type as src.
  - **ksize** – Gaussian kernel size. ksize.width and ksize.height can differ but they both must be positive and odd. Or, they can be zero's and then they are computed from sigma .
  - **sigmaX** – Gaussian kernel standard deviation in X direction.
  - **sigmaY** – Gaussian kernel standard deviation in Y direction; if sigmaY is zero, it is set to be equal to sigmaX, if both sigmas are zeros, they are computed from ksize.width and ksize.height ,to fully control the result regardless of possible future modifications of all this semantics, it is recommended to specify all of ksize, sigmaX, and sigmaY.
  - **borderType** – pixel extrapolation method (see borderInterpolate() for details).[11]
- Parameters:**

Next Hough transform is used to detect all the circles in the mask. The Hough transform is a feature extraction technique used in image analysis, computer vision, and digital image processing. The purpose of the technique is to find imperfect instances of objects within a certain class of shapes by a voting procedure. This voting procedure is carried out in a parameter space, from which object candidates are obtained as local maxima in a so-called accumulator space that is explicitly constructed by the algorithm for computing the Hough transform. The classical Hough transform was concerned with the identification of lines in the image, but later the Hough transform has been extended to identifying positions of arbitrary shapes, most commonly circles or ellipses [12].

```
void HoughCircles(InputArray image, OutputArray circles, int method, double dp,
double minDist, double param1=100, double param2=100, int minRadius=0, int
maxRadius=0 )
```

- **image** – 8-bit, single-channel, grayscale input image.
  - **circles** – Output vector of found circles. Each vector is encoded as a 3-element floating-point vector.
  - **circle\_storage** – In C function this is a memory storage that will contain the output sequence of found circles.
  - **method** – Detection method to use. Currently, the only implemented method is CV\_HOUGH\_GRADIENT.
  - **dp** – Inverse ratio of the accumulator resolution to the image resolution. For example, if dp=1 , the accumulator has the same resolution as the input image. If dp=2 , the accumulator has half as big width and height.
  - **minDist** – Minimum distance between the centers of the detected circles. If the parameter is too small, multiple neighbor circles may be
- Parameters:**

falsely detected in addition to a true one. If it is too large, some circles may be missed.

- **param1** – First method-specific parameter. In case of `CV_HOUGH_GRADIENT`, it is the higher threshold of the two passed to the `Canny()` edge detector (the lower one is twice smaller).
- **param2** – Second method-specific parameter. In case of `CV_HOUGH_GRADIENT`, it is the accumulator threshold for the circle centers at the detection stage. The smaller it is, the more false circles may be detected. Circles, corresponding to the larger accumulator values, will be returned first.
- **minRadius** – Minimum circle radius.
- **maxRadius** – Maximum circle radius.[13]

The following loop draws circumferences around all detected circles. This is how the tracking works.

From this point there is image analysis connected to the drone movement. The algorithm works that way:

- if the ball is too close (circumference of the circle is bigger than 40 units) to the drone, it goes back, if it is too far (circumference of the circle is less than 20 units) the drone goes forward
- if the distance is appropriate (around 40cm), then the vertical position is being analyzed and the drone moves up or down
- if the vertical position is appropriate (center), then the horizontal position is being analyzed and the drone moves left or right.

The last part is the main function. `Ros::init()` function must be called before using any other part of the ROS system. It needs to see `argc` and `argv` so that it can perform any ROS arguments. `NodeHandle` is the main access point to communications with the ROS system. The first `NodeHandle` constructed will fully initialize this node, and the last `NodeHandle` destructed will close down the node. Then the definitions of `topic_takeoff`, `topic_landing` and `msg_vel` are presented. They are used in movement functions when the drone is taking off, landing or the velocity is being changed and it is needed to publish one of these topics. Calling `ardrone/flatrim` inside service without any parameter will send a "Flat Trim" request to AR-Drone to re-calibrate its rotation estimates assuming that it is on a flat surface. Do not call this service while drone is flying or while the drone is not actually on a flat surface. The drone takes off. `Ros::init()` function has to be called once again, because we are going to use `ImageConverter` class. Then the object of that class is being created and the callback is being executed in a loop. Additionally if sb pushes escape button for 30ms the program ends.

```

class ImageConverter
{
    ros::NodeHandle nh_;
    image_transport::ImageTransport it_;
    image_transport::Subscriber image_sub_;
    image_transport::Subscriber image_sub_depth_;
    image_transport::Subscriber image_sub_threshold;
    image_transport::Publisher image_pub_;
    image_transport::Publisher image_pub_depth;
    image_transport::Publisher image_pub_threshold;

public:
    ImageConverter()
        : it_(nh_)
    {
        image_sub_ = it_.subscribe("/camera/rgb/image_raw", 1,
            &ImageConverter::imageCb, this);
        cv::namedWindow(OPENCV_WINDOW);
        image_pub_ = it_.advertise("/image_converter/output_video", 1);
    }

    ~ImageConverter()
    {
        cv::destroyWindow(OPENCV_WINDOW);
    }

    void imageCb(const sensor_msgs::ImageConstPtr& msg)
    {
        cv_bridge::CvImagePtr cv_ptr;
        try
        {
            cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
        }

        catch (cv_bridge::Exception& e)
        {
            ROS_ERROR("cv_bridge exception: %s", e.what());
            return;
        }

        //-----
        cv::Mat HSVImage;
        cv::cvtColor(cv_ptr->image, HSVImage, CV_BGR2HSV); //HSV
        cv::Size size = HSVImage.size();
        cv::Mat mask = cvCreateMat(size.height, size.width, CV_8UC1); //MASK
        cv::inRange(HSVImage, cv::Scalar(1,115, 119), cv::Scalar(8,255,255), mask); //detecting colors

        //morphological closing (fill small holes in the foreground)
        dilate( mask, mask, getStructuringElement(MORPH_RECT, Size(21, 21)) );
        erode(mask, mask, getStructuringElement(MORPH_RECT, Size(10, 10)) );

        //opening
        erode(mask, mask, getStructuringElement(MORPH_RECT, Size(11,11)) );
        dilate( mask, mask, getStructuringElement(MORPH_RECT, Size(5, 5)) );

        GaussianBlur( mask, mask, Size(15, 15), 2, 2 );
        vector<Vec3f> circles;
        HoughCircles( mask, circles, CV_HOUGH_GRADIENT, 1.5, size.height/10, 100, 40, 0, 0); // hough transform

        float x,y,r;
        for( size_t i = 0; i < circles.size(); i++ )
        {
            Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
            int radius = cvRound(circles[i][2]);
            // circle center
            circle(mask, center, 3, Scalar(0,255,0), -1, 8, 0 );
            // circle outline
            circle(mask, center, radius, Scalar(0,0,255), 3, 8, 0 );
            // circle center
            circle(cv_ptr->image, center, 3, Scalar(0,255,0), -1, 8, 0 );
            // circle outline
            circle(cv_ptr->image, center, radius, Scalar(0,0,255), 3, 8, 0 );
            x=circles[i][0];
            y=circles[i][1];
            r=circles[i][2];
        }
    }
}

```

```

    r=circles[i][2];
}
    if(circles.size()>0)
    {
        printf ("Wspolrzedna x: %f \n  Wspolrzedna y:  %f  \n  Promien: %f \n Ilosc kolek: %d \n", x, y,r,circles.size());
    }
    bool temp1,temp2; //help variables
    if(circles.size()==0)
    {
        stop();
    }
    if(r<20 && circles.size()>0)
    {
        forwardx();
        temp1=TRUE;
    }
    if(r>40)
    {
        backx();
        temp1=TRUE;
    }
    if(r<40 && r>20)
    {
        if(temp1==TRUE)
        {
            stop();
            temp1=FALSE;
        }
        if(y<200 && circles.size()>0) //go up
        {
            up();
            temp2=TRUE;
        }
        if(y>300 && circles.size()>0) // go down
        {
            down();
            temp2=TRUE;
        }
        if(y<300 && y>200)
        {
            if(temp2==TRUE)
            {
                stop();
                temp2=FALSE;
            }

            if(x>350 && circles.size()>0)
            {
                backy();
            }
            if(x<250 && circles.size()>0)
            {
                forwardy();
            }
            if(x<350 && x>250)
            {
                stop();
            }
        }
    }
}
cv::imshow(OPENCV_WINDOW, cv_ptr->image); //show detected ball
};

```

Picture 10 - main.cpp, image conversion and control algorithm.

```

int main(int argc, char** argv)
{
    //-----DRONE PART-----
    ros::init(argc, argv, "drone");
    ros::NodeHandle n;
    topic_takeoff = n.advertise<std_msgs::Empty>("/ardrone/takeoff",1, true);
    topic_landing = n.advertise<std_msgs::Empty>("/ardrone/land",1, true);
    cmd_vel = n.advertise<geometry_msgs::Twist>("/cmd_vel",1, true);
    service_flattrim = n.serviceClient<std_srvs::Empty>("/ardrone/flattrim");
    adjust();
    printf("Calibration\n");
    takeoff();

    //-----IMAGE PART-----
    ros::init(argc, argv, "image_converter");
    ImageConverter ic;
    while(1)
    {
        ros::spinOnce();

        if (waitKey(30) == 27) //wait for 'esc' key press for 30ms. If 'esc' key is pressed, break loop
        {
            cout << "esc key is pressed by user" << endl;
            break;
        }
    }
    return 0;
}

```

Picture 11 - main.cpp, main function.

## 6. Results of work

As it is presented in the picture below, an orange ball is being detected and position of the ball stimulates the direction of drone's movement. The results are presented in screenshots below. In the first picture the drone is flying up, because the ball is in upper position, analogically the drone flies down when the ball is in lower position.



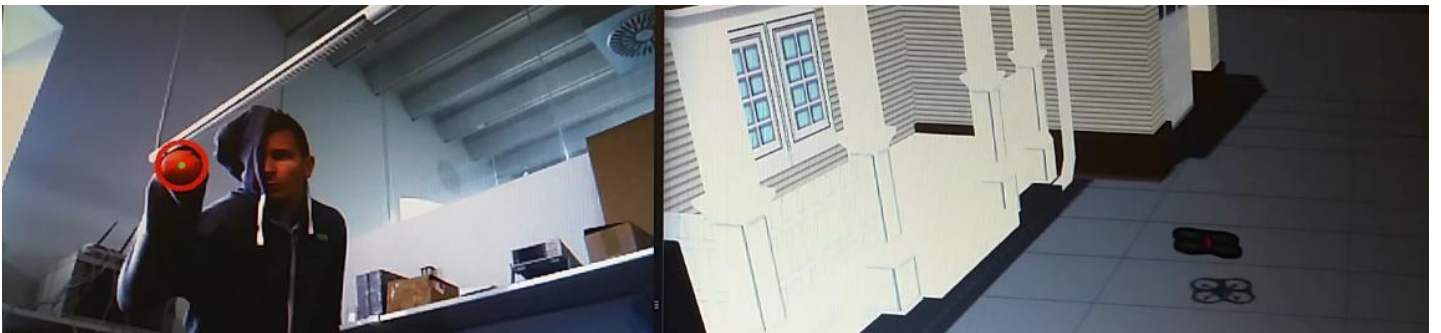
Picture 12 - drone goes up.

In the second picture the drone is flying backwards because ball is close to the camera. Analogically the drone flies forwards when ball is far away.



*Picture 13 - drone goes back.*

In the third picture the drone goes left because the ball is on the left side of the camera. Analogically the drone flies right when the ball is on the right side of the camera.



*Picture 14 - drone goes left.*

## **7. Summary**

To sum up, the target has been reached and the drone is following the ball. There were little problem with the HSV transform and adjusting the color of the ball. In the end it detects also human skin so thats why the person moving the ball should be covered with clothes. As we can see ROS and OpenCV have very wide range of applications and are very helpful in robotics. Huge amounts of functions in these libraries allows the user to experiment with many different objects and to create a

device that will manage many difficult tasks. A very big advantage of such type of projects is that it can be modified in many different ways, which could make the drone much more independent and "intelligent" device.

## 8. Bibliography

[1] <http://wiki.ros.org/indigo/Installation/Ubuntu>

[2] <http://ardrone-autonomy.readthedocs.io/en/latest/installation.html>

[3] [http://gazebosim.org/tutorials?tut=ros\\_installing](http://gazebosim.org/tutorials?tut=ros_installing)

[4] [http://gazebosim.org/tutorials?cat=guided\\_b&tut=guided\\_b2](http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b2)

[5] [http://wiki.ros.org/tum\\_simulator](http://wiki.ros.org/tum_simulator)

[6] [http://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous\\_transformations.html](http://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html)

[7] <http://desktoppub.about.com/od/glossary/g/HSV.htm>

[8] [http://docs.opencv.org/2.4/modules/core/doc/old\\_basic\\_structures.html](http://docs.opencv.org/2.4/modules/core/doc/old_basic_structures.html)

[9] [http://docs.opencv.org/2.4/modules/core/doc/operations\\_on\\_arrays.html](http://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html)

[10] [http://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion\\_dilatation/erosion\\_dilatation.html](http://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html)

[11] <http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html?highlight=dilate>

[12] [https://en.wikipedia.org/wiki/Hough\\_transform](https://en.wikipedia.org/wiki/Hough_transform)

[13] [http://docs.opencv.org/2.4/modules/imgproc/doc/feature\\_detection.html?highlight=houghcircles](http://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=houghcircles)