# Robotics 2016

Juan Pedro Torres Muñoz

***Abstract*— Surveillance task are really dangerous and exhausting, because of this the purpose of this course is to implement a robot that is capable of carrying out a round of surveillance throughout the floor. Making the robot route on the building be configurable, and the robot completely autonomous.**

## I. INTRODUCTION

Through this documentation will be explain all the different steps from the beginning to the final version of the vigilant, the mathematical bases, programming techniques and more information for each delivery, also the possible improvements and fails. During this documentation there will be four sections each one center on one big step on the creation of the final system, the autonomous surveillance robot. The first section is about the controller component capable of move in a room avoiding collisions, the second section will be a improvement of the first section, a robot capable of go to a destination point without collide with any object. The third section will be about a supervisor component capable of send instructions to the robot and move using the AprilTags technology through the room. The last section, will be the final system of the surveillance robot moving through the building.

## II. COMPONENT GOTO X

This delivery consist of a robot capable of moving through a 2D map which has obstacles in it, to a destination point of our election.

For this objective will be use the different components from robocomp, including robocomdsl and RCISMousePicker. It will be used a publish/subscriber model to get the selected point in the 2D map, all this managed by the framework “Ice”.

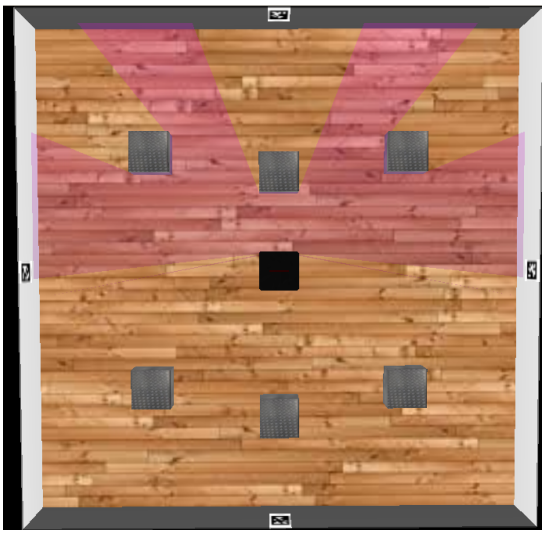


Fig. 1. Map.

### *II-A. IMPLEMENTATION AND SUBSCRIPTION*

The skeleton code of this delivery was made using the robocomdsl code generator tool, using the following file to generate our component.

```
import "/robocomp/interfaces/IDSLs/
    DifferentialRobot.idsl";
import "/robocomp/interfaces/IDSLs/Laser.idsl";
import "/robocomp/interfaces/IDSLs/RCISMousePicker.
    idsl";

Component controller
{
        Communications
        {
                requires DifferentialRobot, Laser;
                subscribesTo RCISMousePicker;
        };
        language Cpp;
        gui Qt(QWidget);
};
```

This code will be use to generate a component which will have a connection to a DifferentialRobot, and a connection to the Laser component, these two connection will be as clients of those components, while also will be subscribe to the RCISMousePicker publications.

Once the file is create, it is necessary to run the next command to create all the sources.

```
robocompdsl controller.cdsl .
```

Now the skeleton it is created, the following step is work with the desire point to move the robot to the point.

### *II-B. CREATION AND STORAGE OF THE POINT*

To work with the received point it is necessary to storage it before use it, so a new structure will be created for this purpose in the "specificworker.h" header file.

```
struct Target
{
  mutable QMutex m;
  QVec pose = QVec::zeros(3);

  float angl;
  bool active = false;

  void setActive(bool newActive)
  {
    QMutexLocker lm(&m);
    active = newActive;
  }

  void copy(float x, float z)
  {
    QMutexLocker lm(&m);
    pose.setItem(0,x);
    pose.setItem(1,0);
    pose.setItem(2,z);
  }

  QVec getPose()
  {
    QMutexLocker lm(&m);
    return pose;
  }
};
```

It will be necesary to implement the subscriber method for the reception of the new values published by the RCIS-MousePicker component. This is the code implemented:

```
void SpecificWorker::setPick ( const Pick &mypick )
{
    pick.copy ( mypick.x,mypick.z );
    pick.setActive ( true );
}
```

The new point will be stored in the previously created structure and marked as a new active point with the next sentence for the algorithm to start.

### *II-C. PROBLEM STATEMENT*

The problem that is presented is the following, the RCIS-MousePicker publish a 2D point on the reference system of the plane, which his own coordinates called (x,z), the robot have to go to this point, but our robot has his own reference system, positioned in the coordinates (xr,zr) and with a angle $\alpha$ representing the rotation from the Z axis of the map plane.

As it can be see, our problem is to obtain the vector which link the actual robot position (xr,zr) with the objective position (x,z),with each point on a different reference system.It will be necessary to change between reference system to calculate the desire vector and rotation angles. This image show the objective.

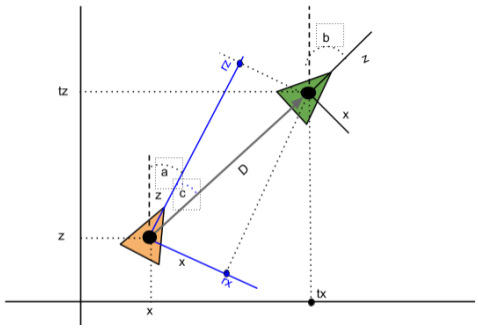



Fig. 2. Objective.

The next mathematical formula can be used to model the problem.

Change reference system equation

$$P_{final} = M * V_r$$

Being $P_{final}$ the coordinates of the objective point in the reference system of the robot and $M$ the following matrix:

$$M = \begin{Bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{Bmatrix}$$

$V_{robotpoint}$ is the vector between the objective point and the robot on the plane reference system. Doing the next calculations, we obtain the vector on the reference system of the robot.

Now we have to apply the next operation and we will obtain the rotation angle to focus the objective point.

$$\arctan(y_{final}, x_{final})$$

The robot will move using the following equations:

$$V_{adv} = distance * 0{,}5$$

The robot once it is looking to the destination, will advance at a speed equal to half of the distance to the objective point. This can be change, and will be change in future versions to other method to calculate the speed. When the robot is at a distance equal to half the "threshold" the robot will stop and wait for the next point.

For the implementation of the system, we have use the structure declared previously which will store the objective point.

### *II-D. IMPROVEMENTS*

There are several improvements to this delivery that will be performed on the next deliveries.

First of all, changing the speed calculation to a exponential function, it will be used a sigmoid function, to achieve a smooth movement.

Another important improvement is the orientation once the objective point is reached, orientating the robot with the Z axis.

And the most important improvement is the achieve that the movement process dodge all the obstacles on the room, allowing the robot to reach every point of the room without crashing with the walls or the obstacles.

## III. GOTOPOINT COMPONENT

During this delivery the robot will start to avoid collisions with the boxes in the room, while it continue to go to the desires points provides by the RCISMousePicker.

The robot need to avoid the obstacles on the room on its way to the target.

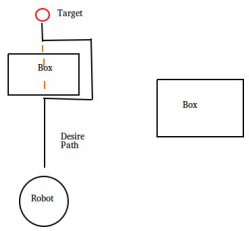



Fig. 3. Robot movement Schema.

The black path represent the path that the robot will follow to reach the target after the implementation of this practice, this movement is divided in two states of the state machine controlling the robot.

Also, there is a orange path, that is the path calculate by the movement algorithm on the previous delivery.

### *III-A. DESCRIPTION OF THE PROBLEM*

For solving the problem it will be necessary to create a state machine in compute method, so this is the structure its going to have.

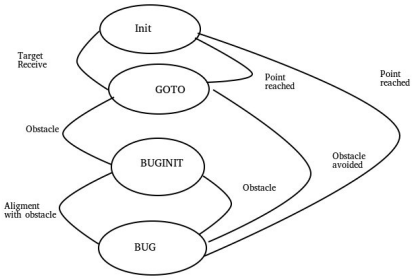



Fig. 4. State Machine of the Controller.

As it can be seen we have four states on our state machine, the first one a initialization state, which do nothing except wait for a target to go for.

The second state is a go state, it simply do all the work from the previous delivery, calculate the rotation and movement to reach the target localization.

The third state is the bug movement initialization, for this delivery the method used to avoid obstacles is a bug movement algorithm, it will be explain later, so this state initialize the algorithm and change to the bug movement.

The last state, bug , it is a state that moves the robot around the obstacles, avoiding any kind of collision, it will be explain in detail ahead.

First of all, when the robot receive a target it start the GOTO state, as the previous delivery, and start to move through the calculated route. Once the robot face a obstacle, store the route calculate, for know when the obstacle has been avoided.

The robot change to BUGINIT state, it start to align with the obstacle saving a prudential distance to avoid any collision at any rotation, the robot rotate at 0.3 rad/s. When it is aligned, it change to the BUG state.

In this state, the robot will advance around the obstacles at a prudential distance and having a own $V_{rot}$ and $V_{advance}$ in each moment.The movement speed and the rotation speed are calculated using the following equations.

$$V_{advance} = 350 * e^{-(|V_{rot}|*\alpha)}$$

$$V_{rot} = \frac{1}{1 + e^{-k*(dist-450)}} - \frac{1}{2}$$

$$\alpha = \frac{\log(0{,}1)}{\log(0{,}3)}$$

The rotation speed ($V_{rot}$) is calculated using the distance of the robot to the obstacle, allowing the robot to rotate slowly when there is a obstacle near. The $k$ factor is a constant set to $0{,}1$ empirically, it is used to soft the slope of the function. This function allow us to have a rotation speed on the interval [-0.5, 0.5].

The advance speed ($V_{advance}$) is calculated using the rotation speed, allowing us to modify the advance speed, while the robot is rotating, making the overall movement of the robot smoother. The $\alpha$ constant is used to modify the absorb for the slope of the curve that calculate the advance speed, it is also set to those values empirically.

### *III-B. COMPONENT BEHAVIOUR*

At the moment, we know how the different speeds are calculated and what does any state, but we still do not know how to change between those states.

INIT state, in this state, the robot do nothing, waiting for a target. Once it receive a target it changes to GOTO state and activates a flag representing that the execution is active. This state also generate and store the line between the target and the robot for future use.

```
case State::INIT:
  if ( pick.active )
  {
    qDebug() << "INIT to GOTO";
        state=State::GOTO;
        ini = QVec::vec3(bState.x, 0, bState.z);
        linea = QLine2D(ini,pick.getPose());
  }
  break;
```

GOTO state, during this state the robot works like the previous delivery, but it also has different triggers for change to other states, as the following.

There is a distance check up to finish in case it reach the target without problem, changing again to INIT. Another test is the call to “obstacle” which is a simple call to a method that check if there is some obstacle on the robot view field, in this case, there will be a change to the BUGINIT state, after that there is a simple absorb method to adjust both speeds.

The BUGINIT state, as already explained, during these state the robot calculate the distance from its position to the line stored at the start, and start to check if there is any obstacle in front of it, in case there is no obstacle is that the robot has finished to align with the obstacle, so it can begin the bug movement. If still have and obstacle, the robot simply rotate at 0.3 rad/s

BUG state, this state is where all the work of this delivery resides. First, here we check if the robot is near the target point, so it has arrive, this situation may occur sometimes.

If the robot has not arrived, after that it check if the robot is half way around the obstacle with a clear view of the target, if this happens, the robot will change to the GOTO state and move in a straight line. If this is not happening, the controller check for obstacles ahead the same way that in the GOTO state.

Following these test, the controller calculates both speeds, and applies those to the robot, after that it checks again if the robot is approaching the original line, the controller will change to the GOTO state, and tries to go to the target from that position, these condition helps in the situation where the robot is moving around a box, but the point is behind another box, as the fig.5 shows.

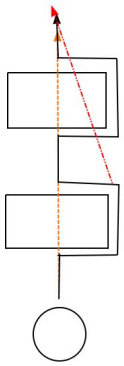
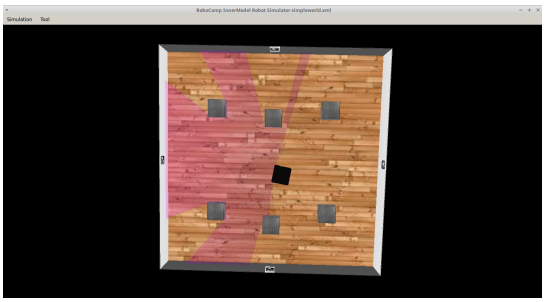


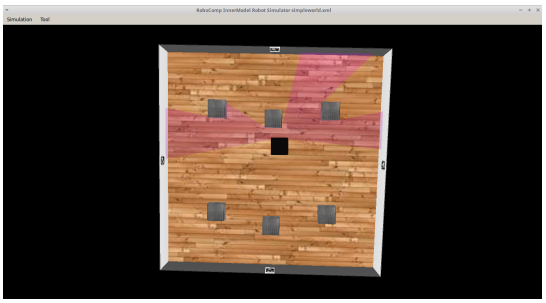
Fig. 5. Change produced to the route.

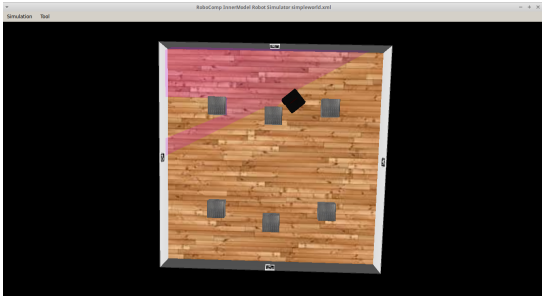
As is explained before, there is a little change in the moment that the controller will change to go to the objective in the case of two boxes followed, being the red arrow the change obtained for the last test on the movement method.

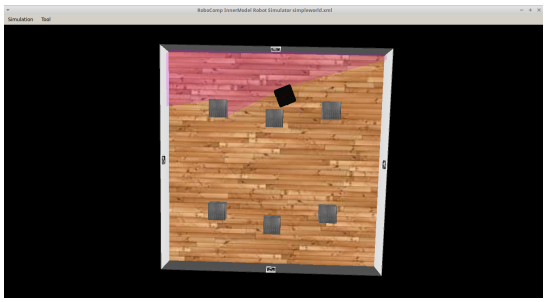
### *III-C. WORKING EXAMPLE*

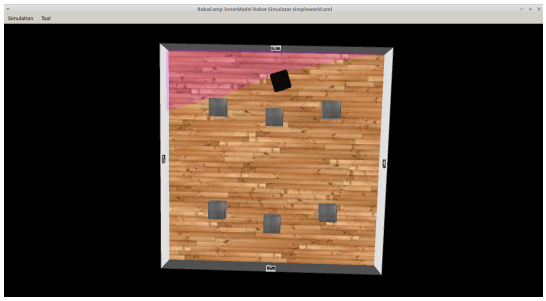
This will be the movement realized in the simpleworld map.











### *III-D. IMPROVEMENTS*

After the development of this delivery we practice the work with state machines and work with a simply algorithm to reach our goal, although the delivery has a lot of working hours, there are several improvements.

One possible improvement is to adjust all the constant used on the algorithm to make smoother the movement of the robot. Other improvement can be the recognition of the obstacle before reaching it, so the move around it can start earlier.

There are for sure more possible improvements to the movement conditions and speed calculations on the different states.

## IV. SIMPLE SUPERVISOR

Currently our robot can move in the room without colliding with the objects in it, and going from one selecting site to another. This is helpful but no the desire objective, in this delivery the robot must move between different tags on the room using the AprilTag component from Robocomp.

For achieve the objective the controller component will communicate with the new component called “Supervisor” that will be create during this delivery. For this delivery the controller component will have to expose some methods, for doing this, it will implement the GotoPoint interface from Robocomp. The Supervisor component will subscribe to the AprilTag component and will require services from the controller. The supervisor, also implements a state machine to control the work.

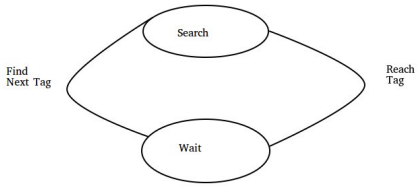



Fig. 6. State Machine of the Supervisor.

These states are very simple and they will be explained in detail later.

The idea of this delivery is that the controller guide the robot from a list of tags dispose in the room, for this objective it will be necessary to use some kind of visual recognition of the tags. All the work on visual recognition is saved on the AprilTag component, so we only have to subscribe to the component and the job is done.

The controller will have to expose some method for the supervisor to be able of communicate and give orders to it. There will be used the GotoPoint interface, which provides methods like: go, stop, turn, etc.

The Supervisor component will have a structure to storage the tags data and the simple state machine describe before.

### *IV-A. IMPLEMENTATION*

First of all, we have to regenerate the controller component using the robocomdsl tool, used in the first practice. The file called “controller.cdsl” have to be edited to :

```
import "/robocomp/interfaces/IDSLs/
    DifferentialRobot.idsl";
import "/robocomp/interfaces/IDSLs/Laser.idsl";
import "/robocomp/interfaces/IDSLs/RCISMousePicker.
    idsl";
import "/robocomp/interfaces/IDSLs/GotoPoint.idsl";

Component controller
{
        Communications
        {
                requires DifferentialRobot, Laser;
                subscribesTo RCISMousePicker;
                implements GotoPoint;
        };
        language Cpp;
        gui Qt(QWidget);
};
```

After the component has been regenerated, we will have to implements the methods of the new interface. They will be implemented as following.

As we can see, there are very simple method, that just modify or start the state machine of the controller.Since the controller was already working properly, we do not have to touch anything else.

Now we have to start to work on the Supervisor, we will generate it using the robocomdsl tool with the following file:

```
import "/robocomp/interfaces/IDSLs/GotoPoint.idsl";
import "/robocomp/interfaces/IDSLs/AprilTags.idsl";
import "/robocomp/interfaces/IDSLs/
    DifferentialRobot.idsl";

Component supervisor
{
        Communications
        {
                requires GotoPoint,
                    DifferentialRobot;
                subscribesTo AprilTags;
        };
        language Cpp;
        gui Qt(QWidget);
};
```

Using this file the generated class will have an empty method to fill for the reception of the published tag from the AprilTag component.

```
void SpecificWorker::newAprilTag(const tagsList &
    tags)
{
    tagDir.Init(innerModel);
    tagDir.copy (tags.front().tx,tags.front().tz,
        tags.front().id);
}
```

It just initializes and stores the new tag in the structure created for this purpose, the structure is the following.

```
struct tagR
{
        mutable QMutex m;
    QVec pose;
    QVec poseAnt = QVec::zeros(3);
    int id;
    InnerModel* model;
    bool hasChanged()
    {
          if( (pose - poseAnt).norm2() > 100)
          {
                poseAnt = pose;
                return true;
          }
           return false;
    };

    void Init(InnerModel* inm)
    {
           model = inm;
    }
    void copy(float x, float z,int id_)
    {
           QMutexLocker lm(&m);
           pose = model->transform ("world",QVec::
               vec3(x,0,z),"rgbd");
           id = id_;
    }
    QVec getPose()
    {
           QMutexLocker lm(&m);
           return pose;
    }
    int getID()
    {
          QMutexLocker lm(&m);
          return id;
    }
};
```

The copy method will transform the coordinates of the tag from the camera reference system to the world coordinates, allowing us to use that coordinates for moving. All the actions with the structure are made in mutual exclusion, because we have to maintain the coherence of the values during all the execution, making sure that there will be no reading when we are receiving a new value. For doing this, we will use a mutex provide for QMutexLocker which provides this

The only thing left is the state machine inside the supervisor, it is the following code.

```
switch(state)
{
  case State::SEARCH:
    qDebug()<< "State SEARCH" ;
    if (tagDir.getID() == current)
    {
      gotopoint_proxy->stop();
          gotopoint_proxy->go("base",tagDir.getPose
              ().x(),tagDir.getPose().z(),0);
          state = State::WAIT;
          qDebug()<<"State change to WAIT";
    }else
          gotopoint_proxy->turn(0.6);
    break;
  case State::WAIT:
        qDebug()<< "State WAIT" ;
        if (gotopoint_proxy->atTarget() == true )
        {
          gotopoint_proxy->stop();
          state = State::SEARCH;
          qDebug()<<"State change to SEARCH";
```

```
            current = (current+1)%4;
        }
    break;
}
```

In the SEARCH state the supervisor checks if the tag the robot is facing is the current tag to go. In case it is the same, the supervisor orders to stop turning and start moving to the point that provides the tag. In case the tag is not the same the supervisor just let the robot rotate until it see the desire tag. This technique works in the current map because the tags are higher than the obstacles and there are no walls.

In the WAIT state the supervisor checks if the robot has reached the target, in case it has reached the target, the supervisor change the current tag and change to the SEARCH state again.

As it can be see, this is a prior approach to the final objective of the robot moving across different rooms on the EPCC map. This is the reason why this practice was so simple, it was just a test to launch and communicate several components.

## V. CONCLUSIONS AND IMPROVEMENTS

During this delivery we only practice the launching and generation of several components as a preparation of the structure for the next practice, doing a simple supervisor for trying the interface GotoPoint.

As this practice is a intermediate step between two very different states of the project there are several things to improve specially on the Supervisor, but in the next delivery we will not use the current supervisor, so it is not necessary to work a lot in the supervisor.

The controller on his part, could have improvements mentioned previously.

## VI. PYTHON SUPERVISOR CONTROLLER

This is the last stage of the work. We are going to change the map, and implements a supervisor able to do the planned task. The new supervisor component will control the robot to perform the surveillance task through the controller component.

First of all, the map has to change.Also we are using a graph library called “Networkx”to allow us to make path between different rooms.

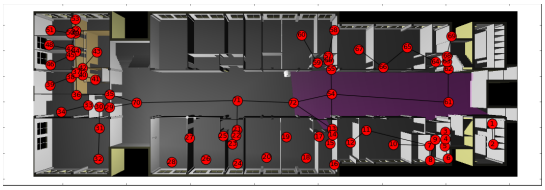


Fig. 7. Final map with graph nodes printed.

This delivery will be implemented in Python unlike all previous. After all the work the robot will be able to visit a list of rooms, using the shortest path between the nodes.

### *VI-A. DESCRIPTION OF THE PROBLEM*

In this delivery the new supervisor has to move the robot for a list of point on the room, that correspond to a list of rooms.

The previous supervisor was coded in C++ and has a simple state machine, so almost nothing of that supervisor can be reused. Once the new supervisor is generated with the robocompdsl tool, we will have to implement the following state machine

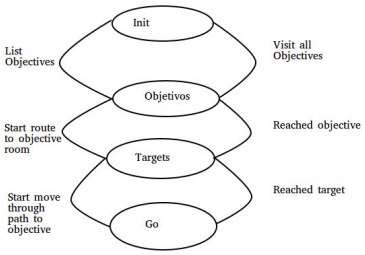



Fig. 8. Supervisor State Machine.

This image son the four states of the new supervisor. It will be explained in detail later. It is necessary for this practice

to install the $networkx$ library to do all the work with the graph. That information will be stored on a text file, and will be loaded each time the component will be launched.

### *VI-B. IMPLEMENTATION*

We will use the following file to generate a new supervisor, using the robocomdsl tool like the previous practices:

```
import "/robocomp/interfaces/IDSLs/GotoPoint.idsl";
import "/robocomp/interfaces/IDSLs/
    DifferentialRobot.idsl";

Component supervisor
{
        Communications
        {
                requires GotoPoint,
                    DifferentialRobot;
        };
        language Python;
        gui Qt(QWidget);
};
```

This will generate the structure of a python component ready to work with the $Ice$ framework and the rest of the components.

Now it will be explained in detail each state of the state machine followed by its implementation.

The first state, INIT just obtain the closest node to the robot, and change to the follow state.

```
        def initState(self):
          print 'SpecificWorker.initState...'
          self.NodoAct = self.nodoCercano()
          self.State += 1
```

The follow state is OBJECTIVE, which will calculate and order the move through the shortest path to reach the objective point as follow:

```
        def Objetivos(self):
          if len(self.ruta) == 0:
            self.State -= 1
            return None
          print 'SpecificWorker.Objetivos...'
          self.NodoAct = self.nodoCercano()
          print self.NodoAct
          self.camino = nx.shortest_path(g,source=
              str(self.NodoAct), target=str(self.
              ruta[0]));
          self.ruta.pop(0)
          print self.ruta
          print self.camino
          self.State += 1
```

As we can see it get the nearest node, calculate the shortest path to the target and change the state to the next. When it goes through all the objectives the state will change to INIT and it will be jumping between INIT and OBJECTIVES.

The TARGET state, will order the movement to the controller between each node of the path to the objective room.

```
        def Targets(self):
          if len(self.camino) == 0:
            self.State -= 1
            return None
          self.nodoObjetivo = self.posiciones[self.
              camino[0]]
          print self.nodoObjetivo
          self.gotopoint_proxy.go("base",int(self.
              nodoObjetivo[0]),int(self.
              nodoObjetivo[1]),0);
          self.camino.pop(0)

          self.State+=1
          return None
```

It will order the movement through to the GotoPoint interface to the controller. Moving the robot for each node on the path to the objective node on the surveillance task. Once it empty the path to the objective node, it change the state to the Objective state to get another path.

The last state the GO state, it will wait for the robot to reach the node ordered to go from the Target state and once it reached it will go back to the Target state, to select the next node to go.

```
        def go(self):
          if (self.gotopoint_proxy.atTarget() ==
              True):
            self.gotopoint_proxy.stop()
            print "HE_LLEGADO"
            self.State -= 1
```

As we can see, the GO state communicate with the controller using the GotoPoint interface.

We have talked about the nearest node, but we have not show code for it, it is the following code.

```
        def nodoCercano(self):
          bState = TBaseState()
          bState = self.differentialrobot_proxy.
              getBaseState()
          r = (bState.x , bState.z)
          dist = lambda r,n: (r[0]-n[0])**2+(r[1]-n
              [1])**2
          #funcion que devuele el nodo mas cercano
              al robot
          return  sorted(list (( n[0] ,dist(n[1],r)
              ) for n in self.posiciones.items() ),
               key=lambda s: s[1])[0][0]
```

For the distance calculation we used a lambda function that allow us to calculate in with a inline call. A lambda function refers to an anonymous function which is not bound to a identifier.

After this, it will order all the values of the list, using the distance value calculate with that lambda function, and returning the node with lower distance. That will generate a problem because, it return the nearest node in a plane world, for example the robot may be near to a node but between the node and the robot there is a wall.
Another task that it is realized each time the components starts is the next.

```
        def readGraph(self):
            self.posiciones = {}
            with open("src/puntos.txt","r") as f:
                for line in f:
                    l=line.strip("\n").split()
                    if l[0]=="N":
                        g.add_node(l[1], x=float(l[2]),y=
                            float(l[3]),name="")
                        self.posiciones[l[1]] = (float(l
                            [2]),float(l[3]))
                    else:
                        g.add_edge(l[1], l[2])
```

This function will read the file called *puntos.txt* on the *src* directory of the component, and using the values on that file it will generate the graph to move around the map.

It is just a loop reading each file of the file which has a fixed structure being the next:

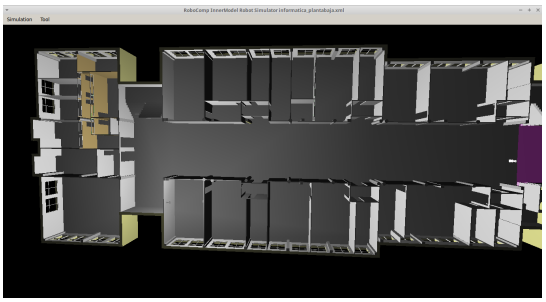
```
Node:
        N id coorX coorY identifierNotUsed
Edge:
        E idNodeA idNodeB
```

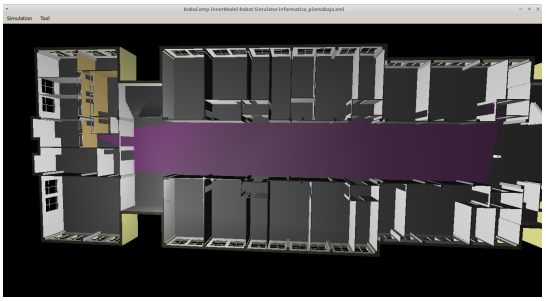
The function will create the graph structure based on that file and create a dictionary with each node id as a key, and the coordinate tuple as the key.

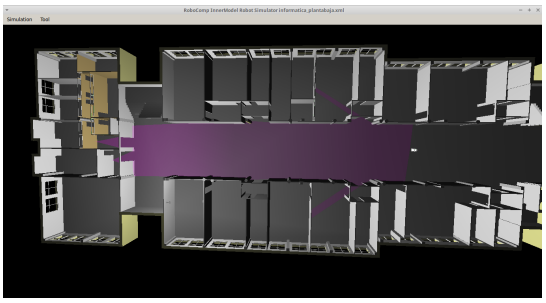
After this work we have a vigilanti robot able to visit a list of rooms from the Computer Science Hall of the EPCC. The robot will visit a prefixed route of rooms, due that at the moment it not possible to add a new route to the robot meanwhile it is moving or after it finished its route. Also, the robot will enter a infinity loop of state change between INIT and OBJECTIVES, due to there is no new route.

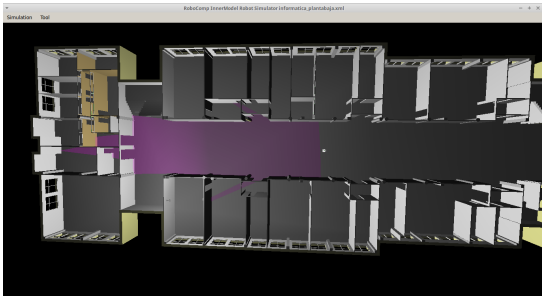
### *VI-C. WORKING EXAMPLE*

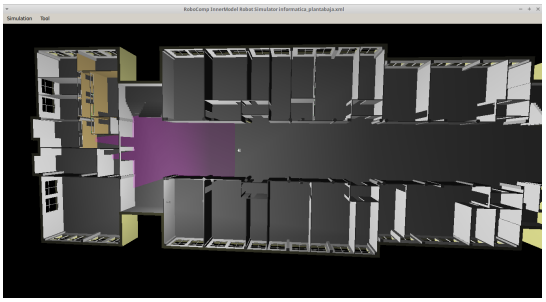
Using the previous show map, we will move the robot from the starting point 61 on the graph to the node 35. These are the images showing the movement.

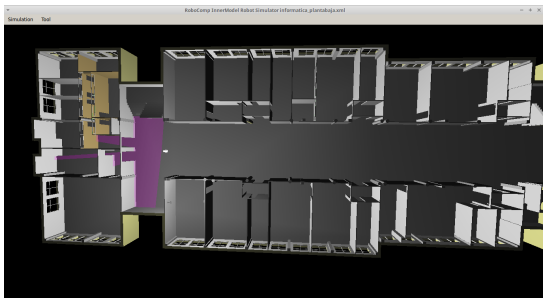












### *VI-D. IMPROVEMENTS*

During the elaboration of the practice we have improve our knowledge about communication between components and our state machine design ability.

Even with everything learned, there are multiple possible improvements in the project, such as:

- Possibility to select a new route to the vigilanti robot to visit.
- Calculation of nearest node does not take into account the walls of the map.
- Many nodes and edges are missing in the graph

These are the most visible improvements of this delivery. We have to take into account that the last deliveries also have possible improvements.

## VII. CONCLUSIONS

From the point of view of the whole practice we have learning about component oriented programming , communication between components, in this case using $Ice$ and we use some code generation tools. Besides all this, we also have practiced the use of Git, using Github as our code repository. Also we have learned a bit about autonomous robots, and movement techniques, and the difficulty on the rotations, advance and the precision required.

## REFERENCES


[1] Component-Based Robotic Engineering, https://campusvirtual.unex.es/zonauex/avuex/pluginfile.php/1077719/mod_resource/content/1/Blocks_-_2009_-_Component-Based_Robotic_Engineering_Part_I_.pdf

[2] Robocomp, https://github.com/robocomp

[3] 2D Transformations, https://docs.google.com/document/d/1FseEHtpTPRrbMCbKonCumc8Vl58UIoCuqDtUU43IBo8/edit

[4] Bug Algorithm, https://www.cs.cmu.edu/~motionplanning/lecture/Chap2-Bug-Alg_howie.pdf

[5] AprilTag: A robust and flexible visual fiducial system, https://campusvirtual.unex.es/zonauex/avuex/pluginfile.php/1339153/mod_resource/content/1/apriltags.pdf

[6] Controller component, https://github.com/JuanPTM/controller

[7] Supervisor component, https://github.com/JuanPTM/Supervisor

[8] Supervisor python component, https://github.com/JuanPTM/supervisorPython

[9] Networkx Graph library, https://networkx.github.io/