Занятие № 3: Базовые пакеты ROS: rviz и tf

rviz

Очень часто, особенно при работе с большим количеством информации, отображения данных в текстовом виде бывает малоинформативным. Так, например, если граф представляется в виде матрицы смежности и имеет размеры 100 на 100, то по текстовому виду этой матрицы довольно сложно представить себе вид этого графа. В таких случаях гораздо удобнее использовать графическое представление данных. Конечно, как любой проект C++, проекты ROS могут использовать графические библиотеки, например, openGL. И, конечно, можно вручную описывать способы представления информации на экране. Но для этих нужд в ROS предусмотрен пакет rviZ, который представляет ноду графического представления (написанную с использованием openGL). Запуск rviz происходит по команде

```
rosrun rviz rviz
```

Окно разделено на три участка. Слева отображаются компоненты, которые могут быть нарисованы. Их довольно много.

Rviz позволяет отображать точки, линии, сетки, объёмные фигуры, направления изменений и многое другое. По центру располагается поле для рисования. Именно здесь появляются изображаемые объекты. Справа располагается настройка Current View. Подразумевается, что rviz будет использоваться для изображения того, что видит робот. А робот имеет свою точку наблюдения. Таким образом можно передавать, например, координаты объектов, помеченные так, как их видит робот, и это координаты будут пересчитаны в абсолютные - те, какими они являются в общем мире на общей карте.

Rviz - обычная нода, и, как и следовало ожидать, она подписывается на топики. Однако, если, ничего не изменяя, выполнить команду

```
rosrun rqt_graph rqt_graph
```

то будет видно, что rviz подписан только на топики /tf и /tf_static. Для того, чтоб rviz читал данные из других топиков, куда будет передаваться информация, в левой части окна rviz необходимо добавить отслеживаемые объекты. Ниже будет продемонстрировано, как отобразить точку.

Итак, отображение точки - это сообщение типа visualization_msgs::Marker. Для того, чтоб rviz был подписан на топик с сообщениями этого типа необходимо в левой части окна представленного на рисунке 6.1, нажав на кнопку Add, выбрать поле Marker. Теперь rqt_graph покажет, что rviz подписан ещё на два топика: /visualisation_marker и /visualisation_marker_array (имя может быть изменено). Теперь в этот топик можно отправлять сообщения. Чем использовать для этих целей команду

```
rostopic pub …
```

Но намного более информативно с точки зрения необходимых заполняемых полей будет создать собственного publisher-а, который будет выводить точку на экран. Ниже представлен листинг программы, которая отображает на экране одну красную точку.

```
 1. #include <ros/ros.h>
 2. #include <visualization_msgs/Marker.h>
 3.
 4. int main(int argc, char **argv) {
 5.  ros::init(argc,argv,"point_publisher");
 6.  ros::NodeHandle nh;
 7.  ros::Publisher pub =
 8.       nh.advertise<visualization_msgs::Marker>("pt_topic",10,true);
 9.  visualization_msgs::Marker point;
10.  point.header.frame_id = "/point_on_map";
11.  point.header.stamp = ros::Time::now();
12.  point.ns = "there_is_point";
13.  point.action = visualization_msgs::Marker::ADD;
14.  point.pose.orientation.w = 1;
15.  point.id = 0;
16.  point.type = visualization_msgs::Marker::POINTS;
17.  point.scale.x = 0.5;
18.  point.scale.y = 0.5;
19.  point.color.r = 1.0;
20.  point.color.g = 0.0;
21.  point.color.b = 0.0;
22.  point.color.a = 1.0;
23.  geometry_msgs::Point p;
24.  p.x = 10;
25.  p.y = 10;
26.  p.z = 5;
27.  point.points.push_back(p);
28.  pub.publish(point);
29.  sleep(1);
30.  return 0;
31. }
```

Разберёмся по порядку, что значат каждые строки в этом коде. В строке 1 подключается общий интерфейс ros. В строке 2 подключается интерфейс использования сообщений типа visualization_msgs::Marker. Как выше было указано, для изображения точки на экран, её необходимо передать для rviz-а именно в формате такого сообщения. Далее в строках 5-8 создаётся топик с именем /pt_topic. Это имя может быть произвольным и его нужно будет указать в rviz. затем на строках 9-27 заполняется сообщение visualization_msgs::Marker. В общем виде visualization_msgs::Marker - это множество точек, которое по-разному интерпретируется с помощью поля класса type. В строке 16 это поле устанавливается в visualization_msgs::Marker::POINTS. Что означает воспринимать все точки, сохранённые в сообщении visualization_msgs::Marker, как просто разрозненный набор точек. Сами точки (которая в примере всего одна) задаётся в строках 23-26 и помещается в сообщение на строке 27. В это сообщение могут быть добавлены ещё точки, и не обязательно создавать новое сообщение, чтобы вывести их на экран.

На строке 10 задаётся frame_id - имя кадра. Это такое имя кадра, которое воспроизводится на ноде rviz. Как было сказано ранее, rviz предоставляет два различных вида: вид общей карты и вид робота. Вот frame_id - это имя, присваемое одному из видов. При запуске rvis необходимо будет связать вид карты с этим именем.

На 11 строке создаётся метка сообщения, обычно эта метка выполняется в виде времени создания этого сообщения.

В 12 строке определяется namespace. Namespace и id (определяемое в 15 строке) однозначно определяют сообщение. Если из двух сообщений поступает информация об объекте с одним и тем же id одного и того же namespace-а, то применяется состояние объекта из последнего сообщения (причём не важно, получено первое сообщение 10 минут или секунду назад, и была ли уже отображена информация на экране или нет).

В 13 строке определяется, что будет сделано с фигурой с указанным namespace-ом и id. Выбран параметр ADD, однако, в случае, если такой объект уже будет существовать, параметр будет выполнять функцию UPDATE.

На 14 строке устанавливается ориентация общего набора точек, передаваемого генерируемым сообщением. Но, так как, всего здесь создаётся только одна точка - изменение этого параметра никак не отобразиться. По параметру point.pose.position с помощью полей x, y, z можно установить координаты (0;0;0) для общего набора, а также с помощью point.pose.orientation повернуть в пространстве.

На строках 17-18 устанавливается толщина точки в местных координатах.

На строках 19-22 устанавливается цвет точек, где a - альфа - параметр прозрачности (1,0 - совсем не прозрачный)

На строке 27 массив точек заполняется единственной точкой.

На строке 28 сгенерированное сообщение посылается в топик.

На строке 29 происходит небольшое ожидание для того, чтоб сообщение не разрушилось деструктором до его помещения в очередь.

В 16 строке можно указать также типы visualization_msgs::Marker::LINE_LIST или visualization_msgs::Marker::LINE_STRIP. LINE_LIST попарно соединяет точки, образуя массив отрезков, в то время как LINE_STRIP просто соединяет точки друг с другом последовательно.

Теперь, как было указано выше, требуется, запустив ноду rviz, указать ему, какой топик слушать и какой frame изменять. Имя frame можно указать в левой части в поле fixed_frame. Название топика же следует указывать в графе Marker. Если такой графы нет, следует добавить с помощью кнопки Add. Имя топика указывается в графе Marker в поле “Marker Topic”. В описанном случае в поле Marker Topic следует указать pt_topic (из строки 8), а в поле “Fixed Frame” указать point_on_map (из строки 10).

Конечно, за одно сообщение можно посылать сразу несколько точек, и не обязательно делать много сообщений такого вида. Однако в этом случае, например, нельзя будет установить цвет каждой точки по отдельности. Для того, чтоб добавить точки для вывода нужно создать больше точек в строках 23-27 и поместить их в вектор .points. А также можно создавать и передавать множество сообщений, а не только одно. Для этого строки 9-27 следует поместить в цикл. Например, учачток кода, который выводит на экран совершающую колебания синусоиду, представлен на листинге ниже.

```
  ros::Rate r(30);
  double offset = 0;
  while(ros::ok()){
```

```
    visualization_msgs::Marker msg;
    msg.header.frame_id = "/point_on_map";
    msg.header.stamp = ros::Time::now();
    msg.ns = "there_is_point";
    msg.action = visualization_msgs::Marker::ADD;
    msg.pose.position.x = 5;
    msg.pose.position.z = -100;
    msg.pose.orientation.x = 100;
    msg.pose.orientation.z = 100;
    msg.id = 0;
    msg.type = visualization_msgs::Marker::POINTS;
    msg.scale.x = 0.5;
    msg.scale.y = 0.5;
    msg.color.r = 1.0;
    msg.color.g = 0.0;
    msg.color.b = 0.0;
    msg.color.a = 1.0;
    for (int x = -20; x <= 20; x++){
      geometry_msgs::Point p;
      p.x = x;
      p.y = 2*sin(x+offset);
      p.z = 2*cos(x+offset);
      msg.points.push_back(p);
    }
    pub.publish(msg);
    offset+=0.4;
    r.sleep();
  }
```

tf

Пакет tf служит для упрощения определения в пространстве координат различных объектов. Например известно, что относительно объекта 1 объект 2 имеет координаты (x,y). А объект 3 относительно объекта 2 - (m,n). Для того, чтобы выяснить взаимное расположение объектов 1 и 3 можно применить формулы высшей математики, однако в этом нет необходимости, поскольку именно для этого предназначен пакет tf.

Разберёмся в механизме работы сразу на примере. Вот, например, исходный код ноды, которая считывает положение черепашки из пакета *turtlesim* и записывает в топик tf. Это действие полезно, поскольку в этом топике информация находится в том виде, в котором её можно будет легко использовать средствами функций, поставляемых в пакете tf.

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <turtlesim/Pose.h>

std::string turtle_name;

void poseCallback(const turtlesim::PoseConstPtr& msg){
  static tf::TransformBroadcaster br;
```

```
  tf::Transform transform;
  transform.setOrigin( tf::Vector3(msg->x, msg->y, 0.0) );
  tf::Quaternion q;
  q.setRPY(0, 0, msg->theta);
  transform.setRotation(q);
  br.sendTransform(tf::StampedTransform(transform, ros::Time::now(),
"world", turtle_name));
}

int main(int argc, char** argv){
  ros::init(argc, argv, "my_tf_broadcaster");
  if (argc != 2){ROS_ERROR("need turtle name as argument"); return -1;};
  turtle_name = argv[1];

  ros::NodeHandle node;
  ros::Subscriber sub = node.subscribe(turtle_name+"/pose", 10,
&poseCallback);

  ros::spin();
  return 0;
}
```

Как понятно из кода, в проргамме создаётся нода, которая подписывается на turtle1/pose (имя взято по-умолчанию). В этот топик нода черепашки пишет координаты черепашки в мире. Как только в этот топик поступает какая-то информация (а поступает она туда постоянно, даже если черепашка стоит на месте), запускается функция, считывающая информацию с этого топика и записывающая её в tf.

Обратите внимание на механизм создания transform: координаты задаются, как члены данные, а поворот при помощи tf:Quaternion.

В функции sendTransform указаны две строковые переменные: «world» и turtle_name. Они будут записаны в сообщение, которое будет отправлено в топик tf. Когда сообщение будет извлекаться, можно будет получить доступ к этим переменным. Их смысловая нагрузка показать, что и в каких координатах считается. В данном случае показано, что в tf отправлены координаты объекта turtle_name относительно world.
Становится понятно, что с помощью такого механизма организации сообщений всегда можно будет восстановить координаты любого объекта, даже если известны лишь его координаты относительно другого объекта, но про тот нам всё известно.
Заметим, что в tf то, относительно чего считаются координаты называется base_frame_id, а то, чьи координты, называется base_frame_id.

Теперь разберёмся, как считывать и обрабатывать сообщения из tf. Рассмотрим, например, такой код:

```
#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <geometry_msgs/Twist.h>
#include <turtlesim/Spawn.h>

int main(int argc, char** argv){
```

```
  ros::init(argc, argv, "my_tf_listener");

  ros::NodeHandle node;

  ros::service::waitForService("spawn");
  ros::ServiceClient add_turtle =
    node.serviceClient<turtlesim::Spawn>("spawn");
  turtlesim::Spawn srv;
  add_turtle.call(srv);

  ros::Publisher turtle_vel =
    node.advertise<geometry_msgs::Twist>("turtle2/cmd_vel", 10);

  tf::TransformListener listener;

  ros::Rate rate(10.0);
  while (node.ok()){
    tf::StampedTransform transform;
    try{
      listener.lookupTransform("/turtle2", "/turtle1",
                               ros::Time(0), transform);
    }
    catch (tf::TransformException &ex) {
      ROS_ERROR("%s",ex.what());
      ros::Duration(1.0).sleep();
      continue;
    }

    geometry_msgs::Twist vel_msg;
    vel_msg.angular.z = 4.0 * atan2(transform.getOrigin().y(),
                                    transform.getOrigin().x());
    vel_msg.linear.x = 0.5 * sqrt(pow(transform.getOrigin().x(), 2) +
                                  pow(transform.getOrigin().y(), 2));
    turtle_vel.publish(vel_msg);

    rate.sleep();
  }
  return 0;
}
```

В программе вызывается сервис «spawn». Таким образом, теперь в turtle_sim_node будет находится две черепашки. Для корректной работы необходимо запустить двух бродкастеров: для turtle1 и для turtle2.

Разберёмся подробно, что делает такой код.
Выходным параметром является направление движения для turtle2, записанное в turtle2/cmd_vel.

Самую важную роль здесь играет функция listener.lookupTransform, которая в переменную transform записывает координаты turtle1 относительно turtle2. Обратите внимание, что в этой функции переменные base_frame_id и child_frame_id стоят в том же порядке, что в

sendTransform: сначала base - относительно чего, а потом child - тот, кого.

Теперь, когда координаты первой черепашки посчитаны (нам не пришлось прилагать усилий для подсчёта этих координат), можно сформировать сообщение и послать в топик, который прослушивает turtle2.

Резюмируя, можно сказать, что tf это очень сильный механизм определения относительных координат объектов. В реальных роботах необходимо ослеживать перемещения десятков движущихся механизмов и конечностей робота. Для того, чтобы легко рассчитывать их взаимное расположение и используется tf.