CSCE574 – Robotics
Spring 2014 – Notes on Images in ROS

1 Images in ROS

In addition to the fake laser scans that we've seen so far with

This document has some details about the image data types provided by ROS, which are much like any other data type, but which also have a few unique quirks.

1.1 Viewing images

The basic data type used to represent images in ROS is called `sensor_msgs/Image`. If you simply want to see the images published on a particular topic of that type, you can use a Camera display in RViz. If RViz seems like overkill—or if rviz has been misbehaving—you could also try a much more lightweight program called `image_view`:

```
rosrun image_view image_view image:=topic_name
```

For example, to see the images being captured by the Turtlebot's Kinect sensor, you might say

```
rosrun image_view image_view image:=/camera/rgb/image_color compressed
```

or

```
rosrun image_view image_view image:=/camera/rgb/image_mono compressed
```

References

- `http://wiki.ros.org/image_view`

1.2 Image Transport

As you might imagine, subscribing to image topics, from a computer other than the one that captures and publishes the images, can consume substantial amounts of network bandwidth. For example, on my system, subscribing to the topic `/camera/rgb/image_color` consumes about 2.5 megabytes per second:

```
$ rostopic bw /camera/rgb/image_color
subscribed to [/camera/rgb/image_color]
average:  2.93MB/s
   mean:  0.92MB min:  0.92MB max:  0.92MB window:  4
average:  2.45MB/s
```

```
      mean:  0.92MB min:  0.92MB max:  0.92MB window:  6
   average:  2.54MB/s
      mean:  0.92MB min:  0.92MB max:  0.92MB window:  9
```

The reason this happens is that those topics contain *uncompressed* video data. Fortunately, ROS provides a means by which we can subscribe to a series of images that are transmitted in a compressed format, and access those messages in an uncompressed format, without worrying about the details of the compression and decompression in our code. The package that provides this feature is called `image_transport`. Some details and tutorials for this package appear in the links below. For now, let's have a look at a minimal example that illustrates how this works:

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>

void callback(const sensor_msgs::ImageConstPtr& msg) {
   ROS_INFO("Got image.");
}

int main(int argc, char **argv) {
   ros::init(argc, argv, "nodeName");
   ros::NodeHandle nh;
   image_transport::ImageTransport it(nh);
   image_transport::Subscriber sub = it.subscribe(
      topicName, queueSize, callback);
   ros::spin();
}
```

Note that, to compile correctly, this example requires a dependency on the `image_transport` package in your `package.xml` and `CMakeLists.txt`.

There is very little difference between this version and the usual ROS topic subscribing you've been doing all semester. However, the `ImageHandler` object will, instead of directly subscribing to the topic you request, first query a string parameter called `~image_transport` to decide whether subscribe to a compressed version of that topic's video stream instead. A few comments:

- In the current version of ROS, there are three valid choices for `~image_transport`:
  - If you leave this parameter unspecified, the `ImageTransport` will subscribe directly to the uncompressed image topic. This is equivalent to not using `image_transport` at all, and is usually a bad idea when the publisher and subscriber run on different computers, especially if there is a wireless link between them.
  - If you set `~image_transport` to `compressed`, each image will be compressed to either a JPEG or PNG image before it is transmitted, and decompressed back to a `sensor_msgs/Image` before your callback is called.
  - If you set `~image_transport` to `theora`, the video stream will be compressed to a very small size using the Theora codec. (See references below.) This tends to be quite a bit more efficient than `compressed`, because it is able to exploit similarities between successive frames.

  The key idea is that you can use run-time parameters to control how the images are transmitted, without writing code that needs to know about the details of the compression that you eventually choose.

- The tilde at the beginning of the parameter name ~`image_transport` indicates that this is a *private name* specific to one node. Perhaps the easiest way to assign such a parameter is to include it between the `node` tags in a launch file. Here's an example:

  ```
  <launch>
     <node name="name" pkg="package" type="type">
        <param name="image_transport" value="theora" />
     </node>
  </launch>
  ```

  More detailed descriptions of private names and private parameters are in Chapters 5 and 7 of the book.

References

- `http://wiki.ros.org/image_transport`
- `http://wiki.ros.org/image_transport/Tutorials`
- `http://wiki.ros.org/image_transport/Tutorials/SubscribingToImages`
- `http://wiki.ros.org/image_transport/Tutorials/ExaminingImagePublisherSubscriber`
- `http://en.wikipedia.org/wiki/Theora`

1.3 OpenCV

You may see references in the ROS documentation to a library called "OpenCV." OpenCV is a free and widely-used library that implements a fairly large collection of machine vision algorithms. Even better, it is already integrated into ROS. We won't try to cover many details about OpenCV in this course—doing so would likely take just as long as learning ROS itself—but you are free to use any OpenCV functions that you find useful. I do suggest, however, using at least a handful of OpenCV functions to access the images that your Turtlebot publishes.

1.3.1 Compiling for OpenCV

To write ROS programs that use OpenCV, the process is somewhat different than for other packages, because OpenCV is not treated as a proper `catkin` package. Instead you'll need to have `cmake` find it separately, using lines like this in your `CMakeLists.txt`:

```
find_package(OpenCV REQUIRED)
target_link_libraries(your-executable-name ${OpenCV_LIBS})
```

You'll also need a dependency on a package called `cv_bridge`, whose job is to convert ROS images into OpenCV images, using the usual `catkin` technique.

1.3.2 Converting images to `cv::Mat`

The most important interest we have in OpenCV is that it provides a data type for accessing images that is somewhat easier to deal with than `sensor_msgs::Image`. Fortunately, it's very easy to convert to the OpenCV image type, which is called `cv::Mat`. For example, you might have an `ImageTransport` callback that contains code something like this:

```
#include <cv_bridge/cv_bridge.h>
#include <opencv/cv.h>
   ...
cv_bridge::CvImagePtr cvImagePtr;
try {
   cvImagePtr = cv_bridge::toCvCopy(msg);
} catch (cv_bridge::Exception &e) {
   ROS_ERROR("cv_bridge exception:  %s", e.what());
}
```

Using the resulting `cv_bridge::CvImagePtr`, we can say `cvImagePtr->image` to get an object of type `cv::Mat`, containing our image data.

1.4 Examining the images

An image is nothing more than a (really big) two dimensional array. If we have a `cv::Mat` (here `Mat` is short for "matrix"), we can determine its size using the data members `rows` and `cols`:

```
cv::Mat &mat = cvImagePtr->image;
int width = mat.cols;
int height = mat.rows;
```

Beyond examining the dimensions of the image, the most basic thing we might want to do is to get the value of one element of that array, say the element at row i and column j. (Notice that the indices are *row* and *column*, not x and y as you might expect. Therefore, $(0, 0)$ is the *top* left of the image, increasing as you move down or to the right.)

There's a method in `cv::Mat` called `at` that can do this, but there is a important complication: `cv::Mat` does not, on its own, know the format (or "encoding") of the image data. That is, it doesn't know whether each pixel is stored as a single byte (as in a monochrome image) or a sequence of three bytes (as in a full-color image) or a floating point number, or something else. As a result, the `at` method is templated by the data type of the individual pixels.

Fortunately, the encodings generated by the Turtlebot are quite predictable, although they do differ for each of the two main image topics.

- Images from `/camera/rgb/image_mono` should have the `mono8` encoding. Each pixel is an unsigned character:

  ```
  cv::Mat &mat = cvImagePtr->image;
  ROS_INFO("image[%d,%d] = %d",
     i, j, (int) mat.at<unsigned char>(i, j));
  ```

  The resulting number can range between 0 and 255, with 0 representing black and 255 representing white.

- Images from `/camera/rgb/image_color` should have the `bgr8` encoding. Each pixel is a collection of three unsigned characters, one for each of red, green, and blue. We can extract all three at once into a `cv::Vec3b` and then access the individual channels one at a time:

  ```
  cv::Mat &mat = cvImagePtr->image;
  cv::Vec3b pixel = mat.at<cv::Vec3b>(i,j);
  ROS_INFO("image[%d,%d] = (r=%d,g=%d,b=%d)",
     i, j, pixel[2], pixel[1], pixel[0]);
  ```

  Note the ordering carefully. In each color channel, the resulting number can range between 0 and 255, with 0 representing no presence of that color at all, and 255 representing full presence of that color.

1.5 Viewing images

One other easily-utilized and potentially useful OpenCV feature is its ability to open GUI windows and display images. In fact, the `image_transport` tutorial on subscribing to images shows an example that does exactly this.

Displaying images is easier than you might think.

- First, you should include the header that declares this GUI functionality:

  ```
  #include <opencv/highgui.h>
  ```

- Second, you should create a window. This probably belongs in your `main` function.

  ```
  cvNamedWindow("windowName", CV_WINDOW_AUTOSIZE);
  ```

  The string name given here is used as an identifier for the window; you should specify the same window name later when you want to display an image.

- Third, also probably in `main`, you should start a separate thread to handle any events the window may need to respond to, such as (very importantly), requests to redraw itself:

  ```
  cvStartWindowThread();
  ```

  You only need to call this function once, even if you create multiple windows.

- Next, when you have a `cv::Mat` image that you want to display (probably in a callback function somewhere), there's a single function call for that:

  ```
  cv::imshow("windowName", mat);
  ```

  This will replace the image currently displayed in that window.

- Finally, when you are completely done using a window, you can close it:

  ```
  cvDestroyWindow("windowName");
  ```

  It is customary (but not strictly necessary) to include this window close operation near the end of your program, after your main loop or `ros::spin()`.

One very powerful way to use this feature is to *modify* the images produced by the robot, before displaying them on your workstation. You can do this by assigning a values to the reference returned by `cv::Mat::at`. For example, you might do something like this to make a pixel black in a `bgr8` image:

```
    cv::Vec3b &pixel = mat.at<cv::Vec3b>(i,j);
    pixel[0] = 0;
    pixel[1] = 0;
    pixel[2] = 0;
```

With some creativity, modifying images can be an extremely powerful technique for gaining insight into what your program is doing.

References

- `http://opencv.org`
- `http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenC`