RQt Image Overlay

Kenji Brameld

Oct 16, 2022

CONTENTS

RQt Image Overlay is an RQt plugin that lets you **easily draw information from topics onto a camera image and view them in RQt**.

GIFs are worth a thousand words!, so:

In case you didn't quite get what's going on, what you see is three layer plugins that are listening to three user-defined msg topics being toggled on and off.

You can effortlessly write your own plugins for RQt Image Overlay to display your custom msg type. All you have to implement is the method that does the drawing, given the msg to draw and the QImage to draw it onto.

CHAPTER

ONE

BACKGROUND

Previously, a common way of achieving something similar was to write a node that subscribes to the topics you want to visualize and the image, draw the topics, and republish the resulting image. This method had several disadvantages:

- Bandwidth being used to republish images
- Inability to toggle layers on/off
- Had to write code to subscribe to the topics and image
- Had to write code tp publish the output image
- Can't easily reuse the code to draw a certain msg type between projects
- Have to recompile to add/remove layers, can't do this during runtime

CHAPTER

TWO

WE'RE HAPPY YOU'RE HERE!

The project is hosted on Github by ROS Sports. **Issues and Pull Requests are welcome!**

2.1 Installation

Follow one of the two installation methods below:

- Binary Installation - For most ordinary users
- Source Installation - For maintainers or cutting-edge users of RQt Image Overlay

> **Warning:** RQt Image Overlay is available for **ROS2 Galactic onwards**. It won't compile on older ROS2 distros and all ROS1 distros.

Binary Installation

Source Installation

Source ROS2, and then run:

```
sudo apt install ros-${ROS_DISTRO}-rqt-image-overlay
```

In your ROS2 workspace, clone the repository and it's dependencies:

```
git clone https://github.com/ros-sports/rqt_image_overlay.git src/rqt_image_overlay --
↪branch ${ROS_DISTRO}
rosdep install --from-paths src
```

In the same directory, build the package and its dependencies by running:

```
colcon build
```

2.2 Usage

> **Warning:** Remember to source your workspace with `. install/local_setup.bash`

2.2.1 Opening

You have the option of Running RQt Image Overlay as either a:

- Standalone Application
- RQt Widget

RQt Widget

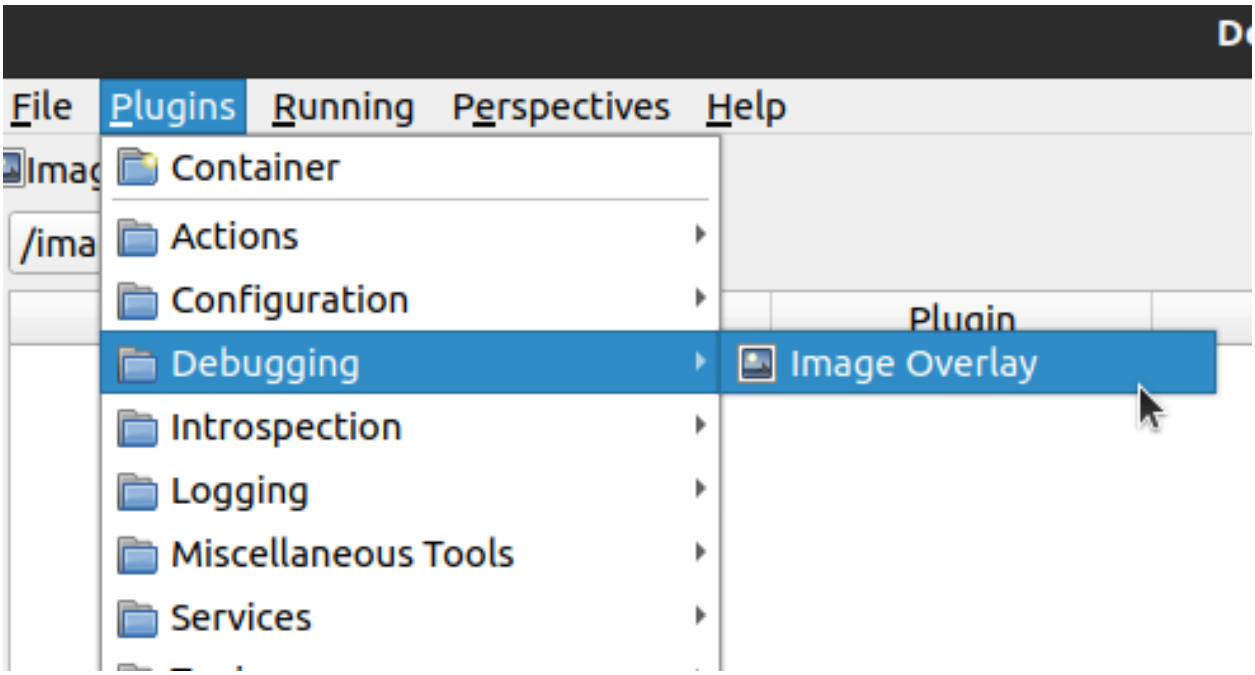
Standalone Application

Open a new terminal and run RQt:

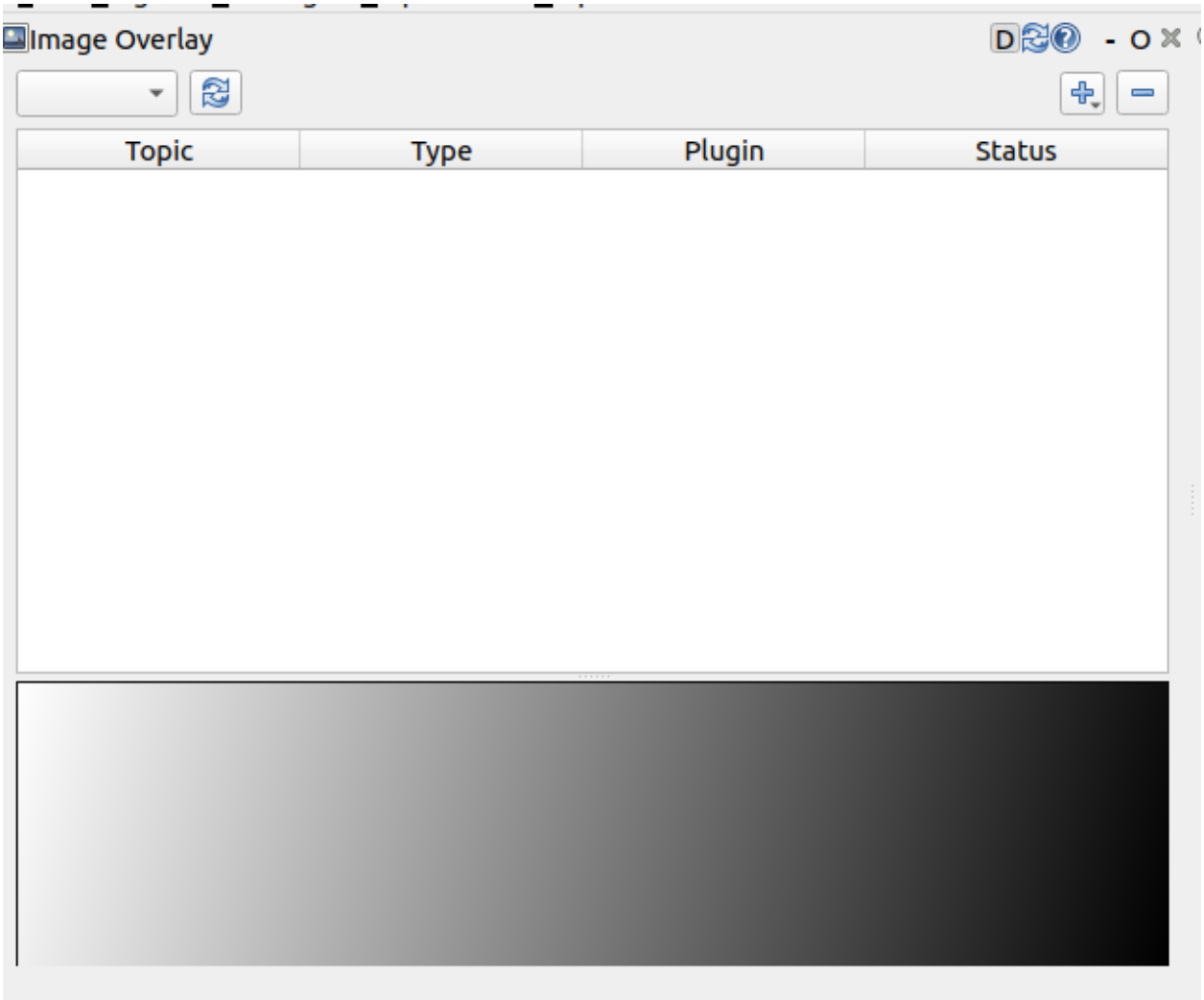
```
rqt
```

> **Warning:** If this is the first time opening rqt since you've sourced the setup file, run `rqt --force-discover` instead.

From the menu bar, select `Plugins` > `Debugging` > `Image Overlay`:



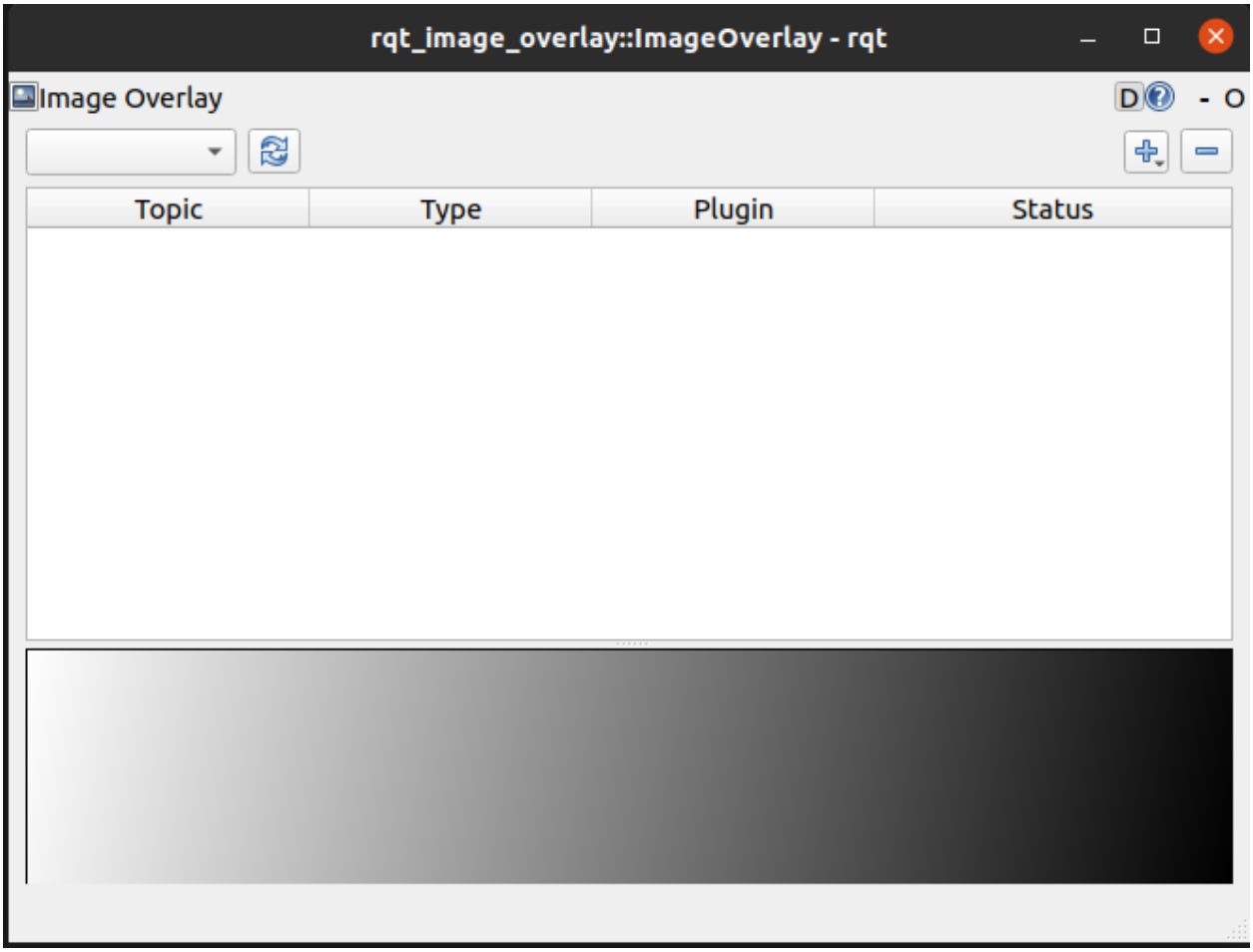
You should see a dockable widget show up like the following:



Open a new terminal and run the application:

```
ros2 run rqt_image_overlay rqt_image_overlay
```

You should see a standalone application show up in RQt like below:



2.2.2 Publishing Images

You need a node that publishes images on a topic to display. Publish images on a topic using **one of the following**:

- V4L2 Node - Recommended, only works if you have a webcam
- Image Publisher Node
- Your own node that publishes images on any topic

V4L2 Node

Image Publisher Node

In this option, you will use the v4l2_camera package to publish images from your webcam. To install the package, run:

```
sudo apt install ros-${ROS_DISTRO}-v4l2-camera
```

Tip: `${ROS_DISTRO}` gets automatically substituted with the name of your ROS2 distro (eg. rolling, galactic, etc.) if you have sourced your ROS2 installation.

To start the v4l2 camera node, run:

```
ros2 run v4l2_camera v4l2_camera_node
```

In this option, you will use the image_publisher package to publish an image file onto a topic. To install the package, run:

```
sudo apt install ros-${ROS_DISTRO}-image-publisher
```

Before starting the image publisher node, you must have an image to publish. In this example, we use an image called test.png in the home directory (ie. `~/test.png`). Replace this with the path to your image file.

To start the image publisher node, run:

```
ros2 run image_publisher image_publisher_node ~/test.png
```

2.2.3 Displaying the Images

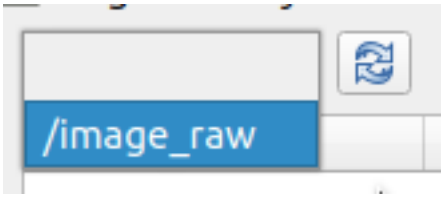
Tip: In a separate terminal, check that the image is being published correctly by running:

```
ros2 topic list -t
```

Make sure `/image_raw [sensor_msgs/msg/Image]` shows up in the list of topics.

Go to the window with the RQt Image Overlay that you opened in the *Opening* section.

Click on the refresh button to update the list of image topics. Opening the drop-down, you should see all topics detected publishing sensor_msgs/Image. In this example, `/image_raw` is the only topic listed.



Select the topic `/image_raw`, you should see the output of your webcam showing in the bottom half of your RQt Image Overlay, as below:

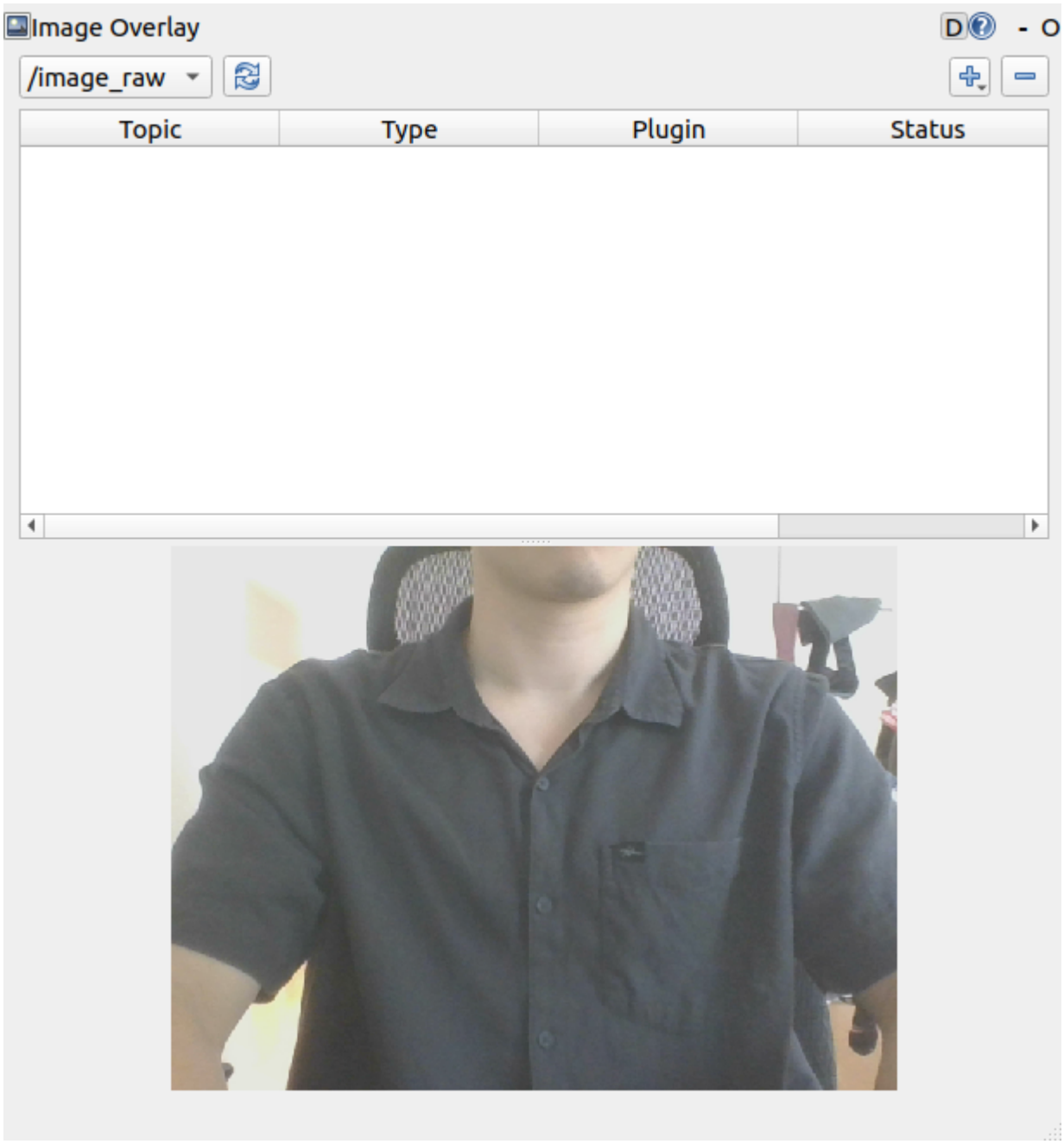


Image Overlay
/image_raw
Topic
Type
Plugin
Status

2.2.4 What's next?

Next, you must create a layer to display your message type. Continue onto *Creating a Layer*.

2.3 Creating a Layer

In this tutorial, you will create a layer that draws a published geometry_msgs::Point onto a published image. The instructions are similar to the Creating and Using Plugins (C++) tutorial.

See also:

Examples for more examples of packages implementing the layer.

2.3.1 1. Create a Package

Create a new empty package in your workspace `src` directory:

```
ros2 pkg create geometry_msgs_layers --build-type ament_cmake --dependencies rqt_image_
↪overlay_layer geometry_msgs --library-name point_layer
```

The command:

- Created a package called geometry_msgs_layers
- Added code to CMakeLists.txt of the new package, to generate a library called *point_layer*
- Generated files `include/geometry_msgs_layers/point_layer.hpp` and `src/point_layer.cpp`

2.3.2 2. Write point_layer.hpp

Open the generated `include/geometry_msgs_layers/point_layer.hpp` file in your favorite editor, and paste the following instead of it:

```
#ifndef GEOMETRY_MSGS_LAYERS__POINT_LAYER_HPP_
#define GEOMETRY_MSGS_LAYERS__POINT_LAYER_HPP_

#include "geometry_msgs_layers/visibility_control.h"
#include "rqt_image_overlay_layer/plugin.hpp"
#include "geometry_msgs/msg/point.hpp"

namespace geometry_msgs_layers
{

class PointLayer : public rqt_image_overlay_layer::Plugin<geometry_msgs::msg::Point>
{
protected:
  void overlay(
    QPainter & painter,
    const geometry_msgs::msg::Point & msg) override;
};

}  // namespace geometry_msgs_layers
```

(continues on next page)

(continued from previous page)

```
#endif  // GEOMETRY_MSGS_LAYERS__POINT_LAYER_HPP_
```

Your PointLayer class must inherit the class rqt_image_overlay_layer::Plugin<T> where T is the msg type you are displaying in the layer (ie. `geometry_msgs::msg::Point`).

2.3.3 3. Write point_layer.cpp

Open the generated `src/point_layer.cpp` file in your favorite editor, and paste the following instead of it:

```
#include <QPainter>
#include "geometry_msgs_layers/point_layer.hpp"

namespace geometry_msgs_layers
{

void PointLayer::overlay(
  QPainter & painter,
  const geometry_msgs::msg::Point & msg)
{
  painter.translate(msg.x, msg.y);

  painter.save();
  QPen pen = painter.pen();
  pen.setCapStyle(Qt::RoundCap);
  pen.setWidth(20);
  painter.setPen(pen);
  painter.drawPoint(0, 0);
  painter.restore();

  painter.translate(-30, -20);
  QString str = "(%1, %2)";
  painter.drawText(0, 0, str.arg(msg.x).arg(msg.y));
}

}  // namespace geometry_msgs_layers

#include "pluginlib/class_list_macros.hpp"

PLUGINLIB_EXPORT_CLASS(geometry_msgs_layers::PointLayer, rqt_image_overlay_
↪layer::PluginInterface)
```

The implementation of point_layer.cpp consists of drawing a black point and drawing the coordinate as text above it.

The arguments to the `PLUGINLIB_EXPORT_CLASS` macro, are:

1. The fully-qualified type of the layer class, in this case, geometry_msgs_layers::PointLayer.
2. The fully-qualified type of the base class, this is always rqt_image_overlay_layer::PluginInterface

---

Important: **The base class is** `rqt_image_overlay_layer::PluginInterface`, which is a non-templated indirect parent class. The direct parent class `rqt_image_overlay_layer::Plugin<T>` cannot be a base class for

plugins because it is a template class.

2.3.4 4. Plugin Declaration XML

A Plugin Declaration XML file must be created to store meta-data about the package.

In the package, create `plugins.xml` with the following code:

```
<library path="point_layer">
  <class type="geometry_msgs_layers::PointLayer" base_class_type="rqt_image_overlay_
↪layer::PluginInterface">
    <description>This is an rqt_image_overlay layer for geometry_msgs::Point</
↪description>
  </class>
</library>
```

Tip: See Plugin Declaration XML from the official ROS2 tutorials to get familiar with the XML tags.

2.3.5 5. CMake Plugin Declaration

CMake Plugin Declaration is required file for the package to be recognised as an rqt_image_overlay_layer plugin.

In your package's CMakeLists.txt, add the `pluginlib_export_plugin_description_file` line after the existing `ament_target_dependencies` line as follows:

```
ament_target_dependencies(
  point_layer
  "rqt_image_overlay_layer"
  "geometry_msgs"
)

pluginlib_export_plugin_description_file(rqt_image_overlay_layer plugins.xml)
```

Important: The first argument to `pluginlib_export_plugin_description_file` (ie. `rqt_image_overlay_layer`) is the plugin category your layer belongs to, not the name of your layer.

2.3.6 6. Build and Run

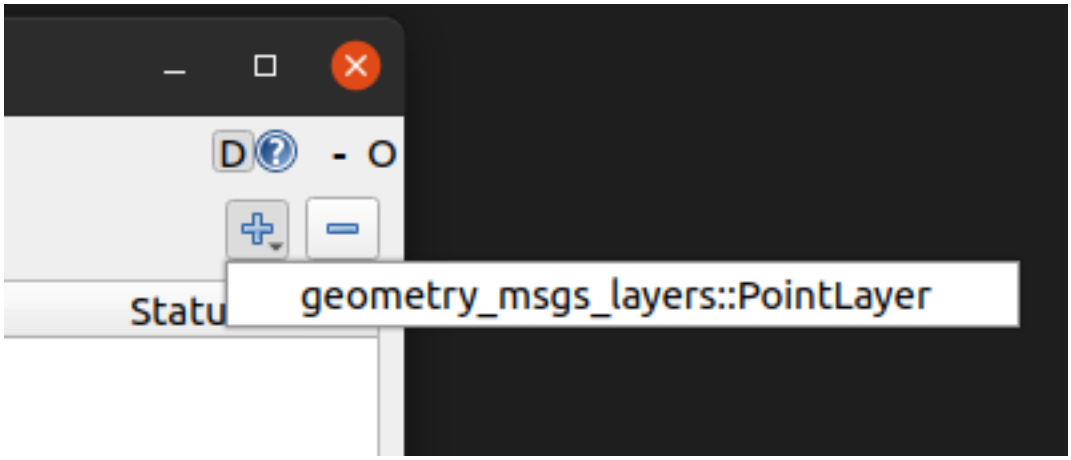
Navigate back to the root of your workspace, and build your new package:

```
colcon build --packages-select geometry_msgs_layers
```

In a new terminal, source your workspace, and either run rqt, or rqt_image_overlay:

```
ros2 run rqt_image_overlay rqt_image_overlay
```

You should be able to see your new layer when you go to add a layer, as following:

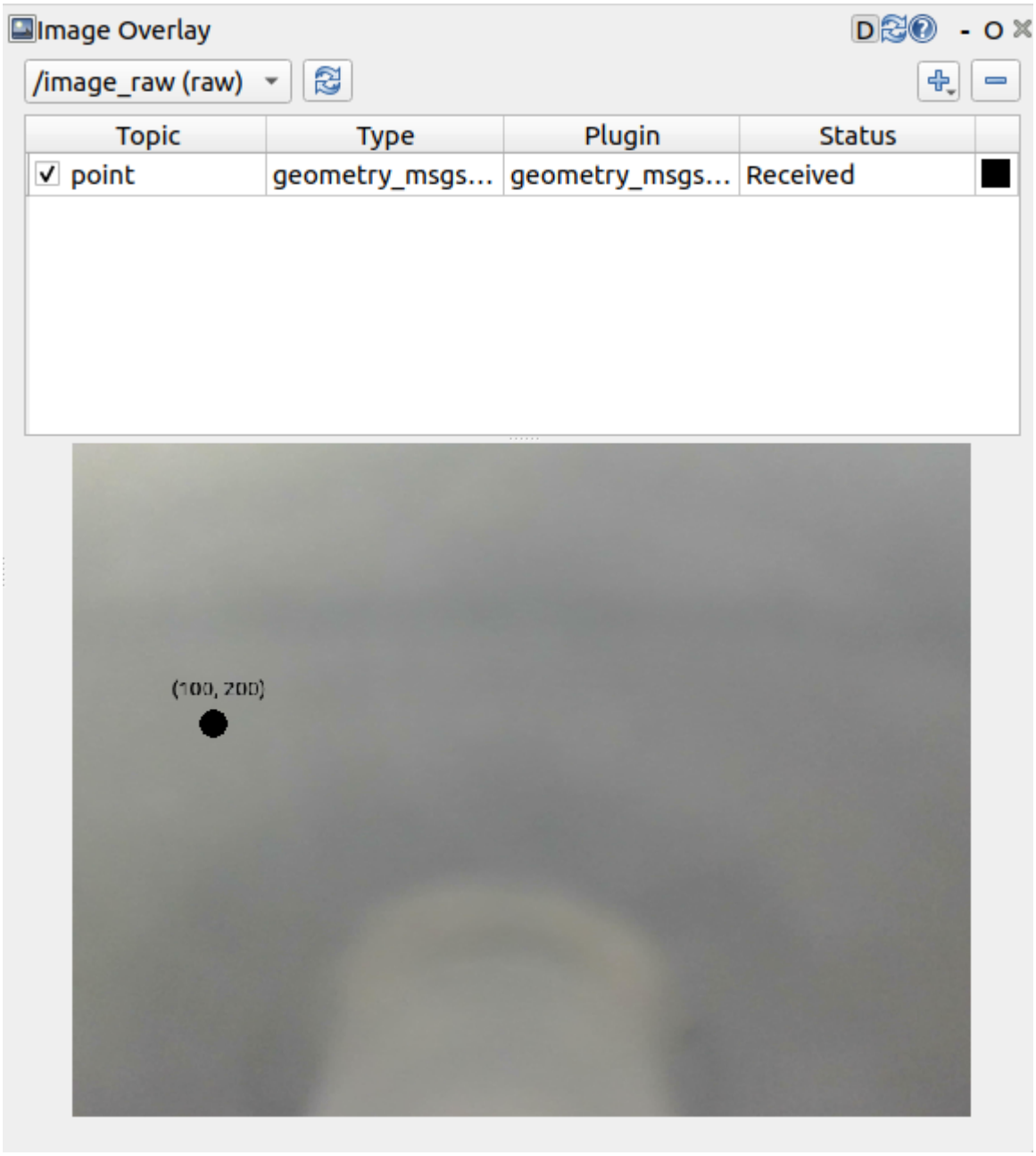


2.3.7 7. Testing

In a fresh terminal, publish a point `(100.0, 200.0)` on topic `point` by running:

```
ros2 topic pub point geometry_msgs/msg/Point "
x: 100.0
y: 200.0
z: 0.0"
```

In rqt_image_overlay, add a geometry_msgs_layer::PointLayer, and set the image topic and set the plugin's topic to `point`. You should see the point layer over the image, as below:



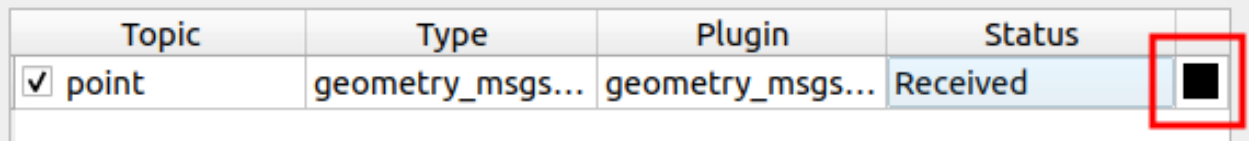
Congratulations! You now know how to visualize any custom ros msg topic onto an image!

2.4 Changing the Layer Color

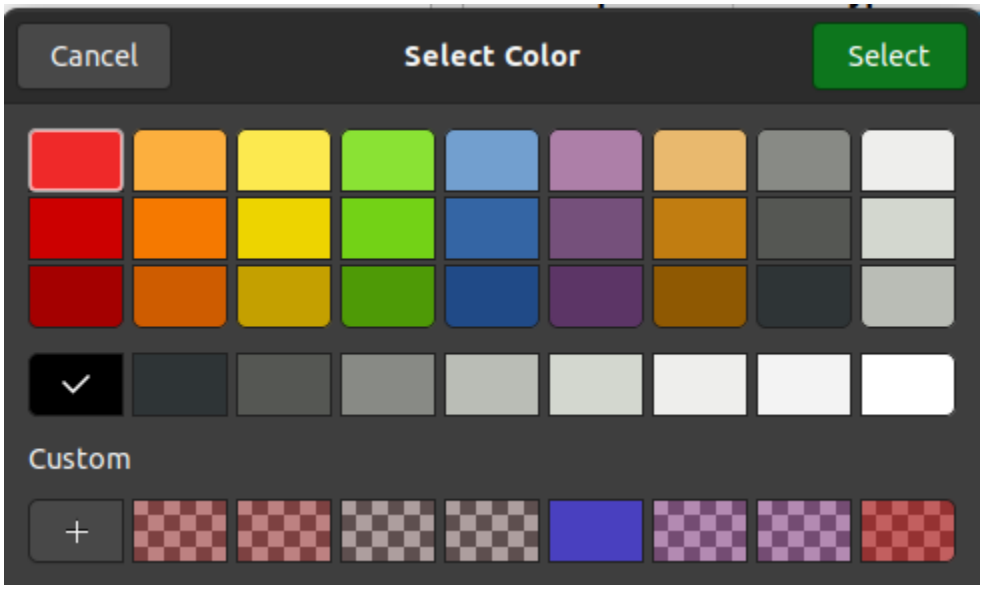
The ability to modify layer colors at runtime is useful for when:

- An overlay is difficult to notice in the image (eg. A green overlay drawn over a green part of an image)
- Two layers are indistinguishable from each other (eg. Two layers using the same plugin type, listening on different topics)

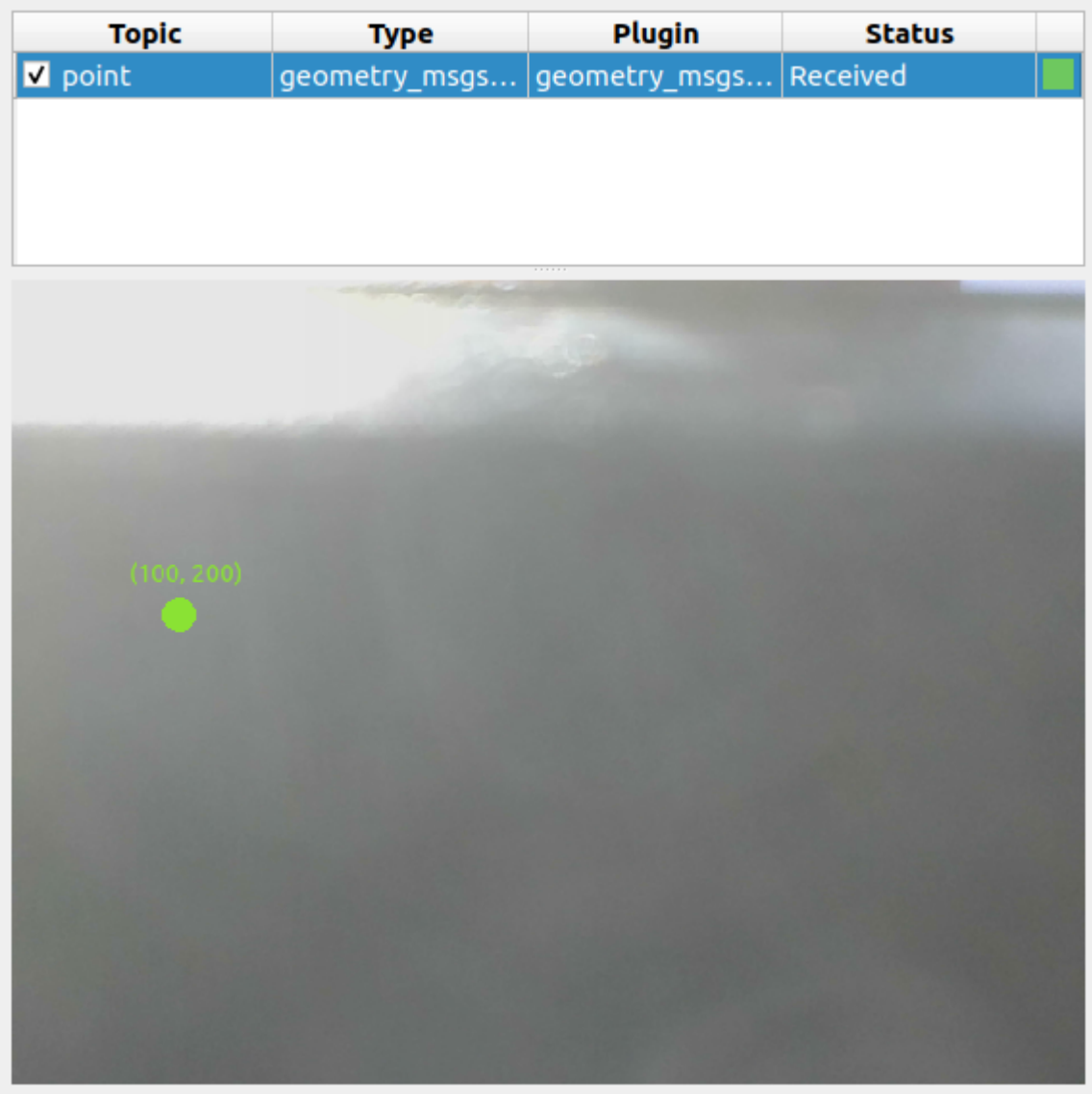
To change the color of a layer, simply **double-click** the color tile in the rightmost column of the layer, as highlighted below:



A color dialog will pop up, where you can select a color, or create your custom color. By default, the layer color is black.



Once you press the Select button to close the dialog, you should see the layer update with the color you selected. In the image below, a light green color was selected:



Tip: The transparency of the layer can also be modified using a custom color.

2.5 Timestamp Synchronization

Timestamp synchronization is the act of synchronizing the layers with the images using header timestamps.

2.5.1 Background

RQt Image Overlay listens for images and messages to overlay on separate topics. Messages on these separate topics arrive at different times. Due to the sources of delay including:

- Image Processing
- Interprocess communication latency
- Network delays

the difference in time that the messages arrive can cause a noticeable lag. The lag can be seen in the following video where the red bounding box around the soccer ball doesn't match the displayed images well:

https://youtu.be/BYN1tczU7xc

With this delay, RQt Image Overlay is almost useless for debugging.

2.5.2 Solution

A solution to this problem, is to use std_msgs/msg/Header to timestamp the messages, such that they match those of the incoming images.

Since messages arrive after the image, we must allow some time for messages to be collected before composing the overlay image and displaying it. This waiting time is called the waiting window, and is **0.3 seconds by default, but can be modified in** *Configurations*.

Note that message_filter's TimeSynchronizer is not used due to its shortcomings:

- Needs to know the msg types at compilation-time as it is templated
- Can only synchronize up to 9 channels
- Reuires all topics to have complete set of matching header timestamps.

If an image is received, and a message for an overlay layer is not received within the waiting window, the overlay is simply not drawn. If for example, you don't detect a certain object in an image, you can not send a message on the topic, and the synchronization will work fine.

The following video shows the ball bounding box matches the images perfectly, when **timestamp synchronization** is being used.

https://youtu.be/1OHLSWl-JzM

Note: Due to this waiting window, images aren't displayed immediately when they are received. They are displayed after the waiting window ends.

Tip: If there are significant delays greater than the waiting window or messages dropping out, RQt Image Overlay can't display them correctly. This can appear as object recognition algorithms failing to recognise objects.

2.5.3 Dealing with messages with no header

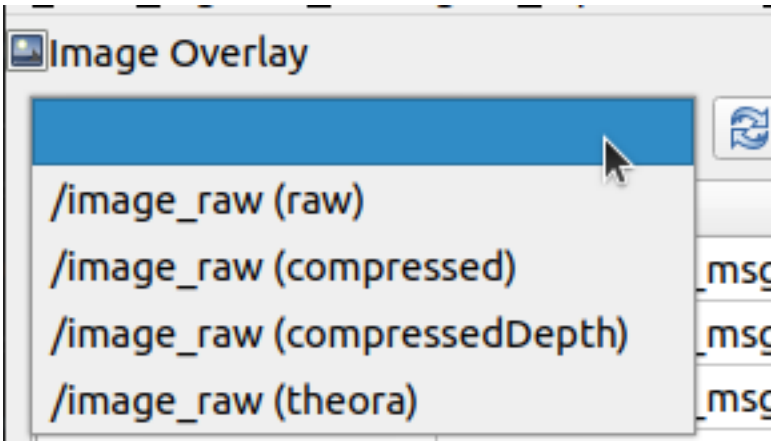
Although having a header field is recommended, RQt Image Overlay can also display messages that don't contain a header. In such cases, the time that the images and messages are received are used to find the nearest msg to display. Note however, that this results in the lag behaviour discussed in *Background* for the corresponding overlay layer however.

2.6 Different Image Transports

Image Transport is used to subscribe to images, which provides transparent support for transporting images in low-bandwidth compressed formats. By using different image transports, you can achieve faster image transportation with possibly a higher frame rate.

To use another image transport, install the image transport on the machine publishing the image, and on the machine running RQt Image Overlay, if they differ.

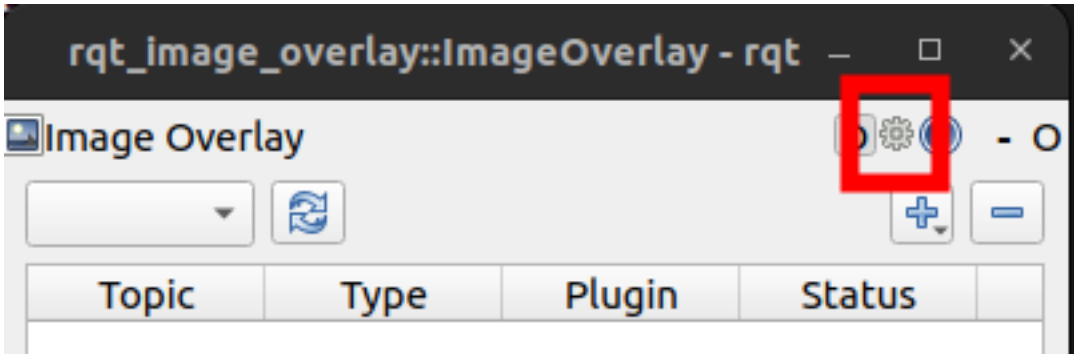
Refresh the image topics in RQt Image Overlay, and you should see alternative transports available, as shown in the image below:



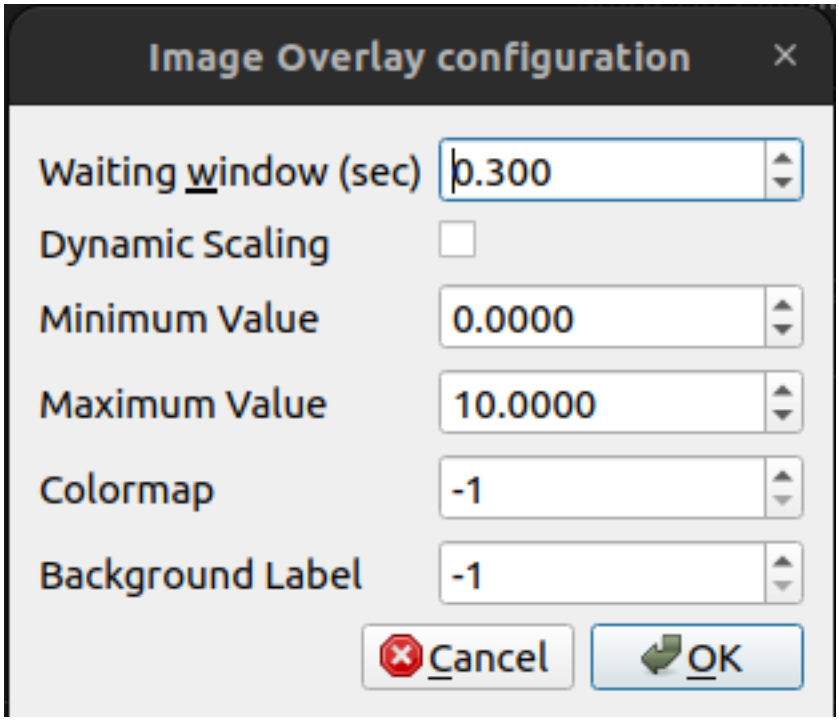
Tip: By using a transport such as *compressed*, you may get better visualization.

2.7 Configurations

Configurations for RQt Image Overlay can be modified through a dialog that is brought up by clicking on the gear icon as shown below:



Clicking on the gear icon will bring up the following dialog:



The settings in the dialog are explained below:

- **Waiting Window (sec)** - The number of seconds to wait before displaying the image overlay in *Timestamp Synchronization*. The default is 0.3s, but you can increase this if you have a large delay between when the image is received and the other messages are received.

The following options only apply to images with a single channel (eg. grayscale images, depth images). Often, such images don't have a range of 0-255, so scaling must be performed to bring the values into the range of 0-255:

- **Dynamic Scaling** - If true, the min/max value in the image is evaluated from the image data, and used to dynamically scale the image. If false, the Minimum Value and Maximum Value below are used to perform non-dynamic scaling.
- **Minimum Value** - Minimum Value used for non-dynamic scaling. Pixels below this value in the original image will be scaled to 0 in the output image.
- **Maximum Value** - Maximum Value used for non-dynamic scaling. Pixels above this value in the original image will be scaled to 255 in the output image.
- **Colormap** - Colormap which the source image is converted with. See ColorMaps in OpenCV for valid values. If set to -1 (default), a colormap will not be applied.
- **Background Label** - Background label when colorizing label image. If set to -1 (default), the background label will be ignored.

2.8 Examples

This page lists example packages from the community that implements the RQt Image Overlay Layer plugin. You should use them as a guide to develop your own!

- apriltag_overlay_plugins - Plugins to display apriltag detections.
- soccer_vision_2d_layers - Plugins to display soccer_vision_2d_msgs.
- vision_msgs_layers - Plugins to display vision_msgs.

If you have a package that you would like to see listed here, please raise a PR by clicking on the top-right *Edit on Github* link!