WebMap - A ROS web interface

Maarten de Vries

November 9, 2012

Contents

Introduction

The goal of the WebMap project was to create a reusable web-interface for safe teleoperation of robots using the Robot Operating System (ROS). The interface consists of a 2D map that represents a top down view of the world, capable of showing multiple robots and their sensor readings. The map is rendered by adding Scalable Vector Graphics (SVG) elements to the web page. The interface also remembers a selected robot, which can be used to display additional information or even control the selected robot. See Figure 1 for a screenshot of the interface with two robots, both equipped with a laserscanner.

Since robots can vary wildly in their capabilities and sensors it is unfeasible to implement them all in the interface. Instead, the map and robots can be extended using a module system, and the robot objects can be used outside the context of a map. This allows users to visualize their robot exactly how they want, and even add extra widgets to the interface that use the same robot objects.

This report starts by giving a general overview of the components in the library in chapter 1. Next, chapter 2 explains some implementation decisions that have to do with rendering the interface and communicating with ROS in more detail. Two tutorials are included to explain how the library should be used in chapter 3, and chapter 4 explains the module system in more detail. Finally, chapter 5 contains a complete API reference.

Some background knowledge regarding the Document Object Model (DOM) and JavaScript is required to read the report, so Appendix A contains a short explanation of both.

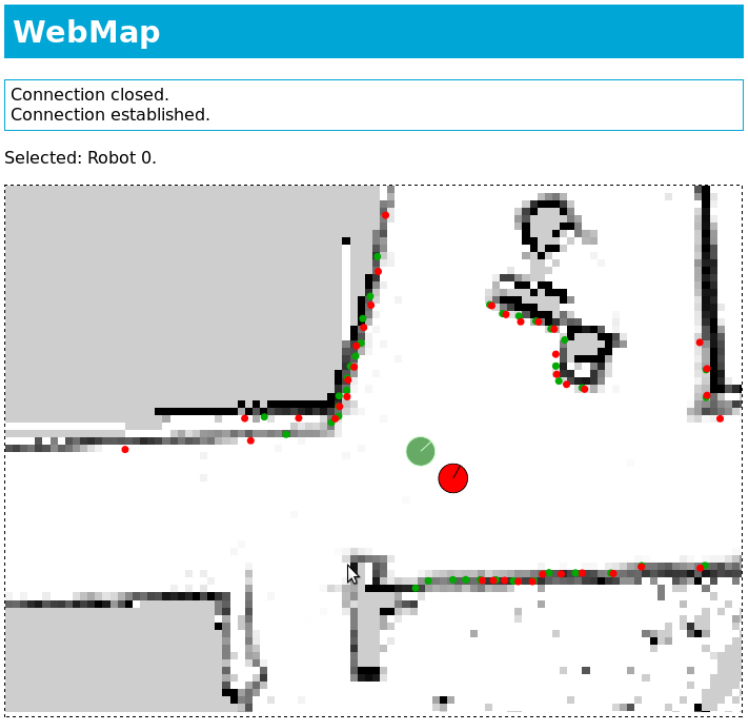


Figure 1: Screenshot of the WebMap interface.

Chapter 1

System overview

This chapter gives a brief overview of the components in the library and the relationships between them.

1.1 Core objects

The heart of the interface is the `Map` object. The `Map` object is responsible for creating and maintaining the root SVG element used to render the map and robots. It also provides several methods to alter the view by translating, rotating or zooming. There are no default key bindings or mouse actions to manipulate the view, but a module is included that provides these.

Once a `Map` object exists, the user can create `Robot` objects and add them to the map for rendering. A `Robot` object maintains an SVG representation of itself within an SVG group element, which a `Map` object can use to render the robot. To retrieve data from the ROS system, the `Robot` objects use `ros.Bridge` objects from the rosbridge library. A `Robot` object can use multiple `Bridge` objects to retrieve information from different ROS systems. `Robot` objects also provide an interface for controlling the robot. One method allows the user to send twist messages to the robot, while another can send goals to a navigation server.

The functionality of both `Map` and `Robot` objects can be changed or extended by adding modules to the respective objects. Both the `Map` objects and the `Robot` objects inherit this functionality from the `Extendable` object. The `Extendable` object has methods to add or remove a module, and to notify all registered modules of an event.

The relationship between the object types is depicted in class diagram in Figure 1.1. While JavaScript is not a class based language, it is possible to use JavaScript prototypes to emulate classical inheritance. For clarity, the methods and properties of the objects have been left out, but they can be found in the API reference in chapter 5.

Note that the diagram shows that both `Map` and `Robot` are `Extendable` objects. Strictly speaking, that means that a `Map` can hold a robot module and vice versa. This would be undesirable (and much more difficult) in a strongly typed language, and would have to be solved using something like C++ templates or Java generics. However, JavaScript is a weakly typed language and doesn't offer type safety in any case, so is no point in creating a more complex hierarchy with separate object types for robot or map modules.

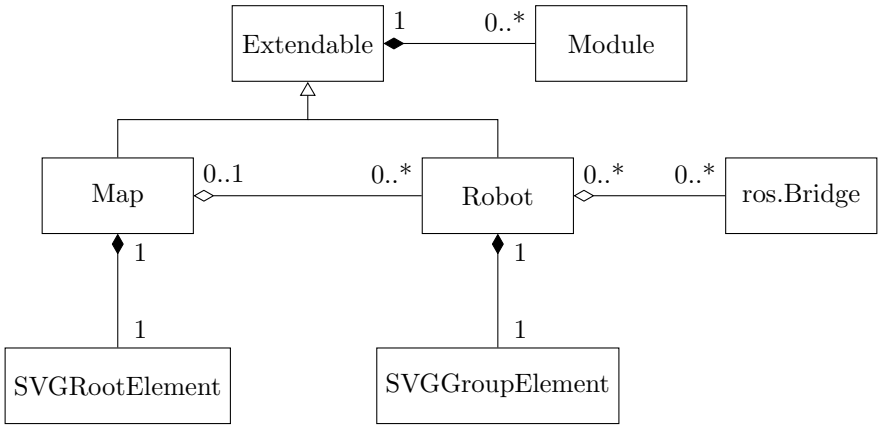


Figure 1.1: Class diagram of the system

1.2 Modules

The library includes three map modules and one robot module. One map module adds mouse controls to change the view of the map, one displays the name of the selected robot and one uses the keyboard to control the selected robot. The included robot module can read and visualizes a laser scanner on the robot.

MouseViewControl The `MouseViewControl` map module enables the user to drag and drop the map in order to translate the view. The scroll wheel is used to zoom the map, or to rotate the map when the shift key is pressed. Additionally, the module captures mouse click events after a drag and drop to prevent other actions from triggering on the click. Otherwise, the end of a drag and drop is also seen as a mouse click and would deselect the currently selected robot, or trigger actions from other modules. The other mouse events, including the mouse up event, will continue to fire normally.

ShowSelected The `ShowSelected` map module shows the name of the currently selected robot inside a HTML element. The entire contents of the element will be replaced by the robot name. The recommended usage is to use a span element with a label or other text outside of the span element, to explain to the user what the information in the span actually means. This module can be used as a template for displaying more detailed information about a robot, or even camera feeds from a robot.

Teleop The `Teleop` map module listens to the W, A, S and D keys and translates them into twist messages. The W and S keys will move the robot respectively forward and backward while the A and D key will rotate the robot counter clockwise and clockwise respectively. It is best suited for differential drives, although it can be used as template for controlling omnidirectional robots. Note however that the module listens to key down and key up events. If, for some reason, the module misses the key up event it will continue to send twist messages to the robot. This could lead to dangerous situations.

LaserScanner The `LaserScanner` robot module can visualize the readings of a laser scanner, and subscribes to a ROS topic to keep the visualization up to date. The module can only read messages of the type `sensor_msgs/LaserScan`. The visual representation consists of a small dot at the location of a valid reading. The dots can by styled by CSS using the following selector: "`.laserscan > .dot`".

Chapter 2

Implementation details

This chapter describes which implementation decisions have been made, and why. The first section explains why SVG was chosen as the rendering technique and how is is being used, while the second section describes how the interface communicates with ROS.

2.1 Rendering

One of the first problems to solve is how to render dynamic graphical content in a web browser. This section compares the most popular techniques with each other and explains why SVG was chosen. Additionally, it will explain how the library uses SVG to actually render a map with robots on it.

2.1.1 Rendering technology

A few years ago, the only cross platform technique for rendering rich dynamic graphics on the web was Flash. At this time though, we got two more options we can use: the HTML5 canvas and SVG. Both technologies have clear advantages over Flash. Firstly, they are supported natively by most browsers whereas Flash requires external software, which is often not available on small devices such as mobile phones. Secondly, both the canvas and SVG can be used from JavaScript, which in contrast to Flash doesn't need to be compiled.

That leaves the questions which of these two technologies to use. The canvas can be used to perform arbitrary draw operations on a bitmap and allows direct pixel access, while SVG enables us to build vector graphics by adding SVG elements to the DOM tree.

Pixel access An advantage of the HTML5 canvas is that is allows direct pixel access, which can be used to apply effects and filters to an image. SVG only has limited support for some pre-defined types of filters. This makes the canvas better suited for some applications, but in the case of this library there is no need for direct pixel access.

Coordinate systems Both the HTML5 canvas and SVG allow us to set up multiple coordinate systems by specifying transformation matrices. This means that we can work in screen coordinates, world coordinates and model coordinates. Screen coordinates can be used to add an overlay over the map to display useful

information, world coordinates are useful for placing robots or other world objects, and finally every robot can have its own coordinate system for placing objects attached to the robot.

Event handlers One clear advantage of SVG is that the added elements can receive events such as mouse clicks. A canvas can also capture mouse clicks, but all draw operations have been flattened into a single bitmap. That means that the only information available is the X and Y coordinates of the click. It is still necessary to deduce which robot (or other object) has been clicked. Depending on the shape of the robots that is not necessarily an easy task. With SVG, the browser does most of the works already and tells us which SVG element has been clicked.

Redrawing Another advantage of SVG is that you can alter the rendering by simply modifying attributes of the SVG elements. Again, the browser does the rest of the work. With a canvas, the only way to update the drawing is by redrawing the rendering or a portion of it from scratch. The canvas can draw anything that SVG can draw, but doing so is less work with SVG.

Styling A final advantage of SVG is that it can by styled by CSS. All draw operations on the canvas have been flattened into a single bitmap so there is no way to apply CSS rules, but the SVG elements remain part of the DOM tree. The effect is that you can style robots differently without using a different model for the robot, and without knowing the desired colors up front in the rendering code. It is even possible to use the CSS selector '`nth-child`' to style robots differently based solely on the order in which they were added.

Taking into account all advantages above, SVG is the technology best suited for creating interactive graphical web-interfaces.

2.1.2 SVG structure

With SVG as the clear winner, the next question is how to use SVG to actually render the map.

The first step is to set up the coordinate systems. With SVG, this is done by creating a group (`g`) element with a transformation matrix. Screen coordinates are already available, so we need to create a world group that transforms world coordinates to screen coordinates. One thing to note is that computer graphics (including SVG) generally use a left-handed coordinate system, while ROS systems generally use a right-handed coordinate system. To compensate, the world transformation matrix starts by flipping the Y axis before scaling, rotating and translating the world. The user can change the scaling, rotation and translation at any time to get a different view of the world.

Next, every robot also gets its own group with a translation and rotation to reflect the pose of the robot. Anything attached to the robot can be added in this group and will automatically move along with the robot.

2.1.3 Rendering robots

Rendering robots sounds simple enough: the proper SVG elements need to be added to the map and updated whenever a robot moves. It must also be done in an efficient manner. Each map is represented by a `Map` object, while each robot is represented by a `Robot` object. The map is responsible for maintaining the root SVG element, while the robot is responsible for generating SVG to draw itself. Furthermore, the robot should also be usable outside the context of the map, to use it with different kinds of widgets.

That presents us with a small problem. If we consider the map to be "just another widget", nothing prevents the user from creating two maps and adding the same robot to both maps. So theoretically, multiple maps

should be able to draw the same robot. However, a single DOM node can only be part of the DOM tree once, and in only one document. That makes it impossible to simply add the same SVG elements to multiple maps. To work around these issues three solutions were considered.

The first option was to create a new Rendering object to represent the link between a robot and a map. This object would maintain a complete SVG representation of the robot. This scheme is depicted in Figure 2.1. Every robot module that adds SVG content to the robot has to be aware of this, because they need to add their SVG content to every rendering of the robot. That also means the robot needs to keep a list of all these rendering objects in case a module has to update its SVG content (due to a new sensor reading, for example). All in all, it is a rather cumbersome method that adds a lot of administrative work.

Figure 2.1: Extra Rendering object.

This can be imporved upon with a different scheme. To sidestep the administrative work of creating, remembering and updating multiple renderings, we can have each robot maintain exactly one SVG representation of itself. Modules can change this representation and don't have to worry about updating more than one rendering. To make sure that multiple maps can still render the same robot, every map must clone the entire SVG content instead of inserting it in the DOM directly. See Figure 2.1 for a diagram of this scheme. It keeps the system simple and it provides the desired many-to-many relationship, but it is not perfect. Every time a robot or module changes the SVG representation, a map has to delete and clone the SVG content again. As you might imagine, this is not a very efficient method since it involves cloning many DOM nodes, potentially many times per second.

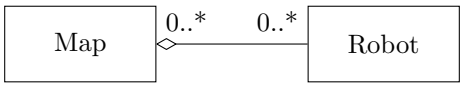


Figure 2.2: Cloning the SVG.

If we recognize that there is actually no sensible use case where multiple maps are displaying the same robot, we can simply ignore the problem altogether and have every robot maintain a single SVG representation. A map does directly insert the SVG content in the DOM tree. To prevent a robot being added to two maps, when a robot is added to a second map it is first removed from the previous map. See Figure 2.3 for a depiction of this scheme. This technique also keeps the system simple, but it doesn't provide the many-to-many relationship anymore. Instead, if a user desperately wants to have multiple maps with the same robot, the user can simply create two robot objects that represent the same physical robot. This has the added benefit that the user can choose to use different modules for the same robot on different maps, or even different models or colors.

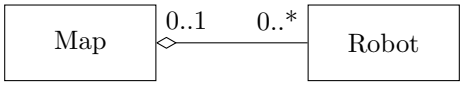


Figure 2.3: Single map per robot.

To prevent the library from becoming overly complex while still keeping good performance, the last method is the one used in the library.

2.2 Communicating with ROS

The communication with the ROS system is done using rosbridge v2.[1] As the name implies it acts as a bridge between a ROS system and non-ROS clients. The rosbridge protocol can use either TCP or WebSockets as transport protocol. A simple JavaScript client for WebSockets capable browsers exists which is ideal for most web-interfaces.

The rosbridge server has to be running on the robot. It is available in the rosbridge_server package in the rosbridge_suite stack.[2].

2.2.1 Connection management

A first thing to note is that a rich web interface might consist of more than just a 2D map, and that it is likely that other widgets or libraries will also have to communicate with ROS. To save resources, anything that can use the same rosbridge connection should do so. That way, if two widgets want to listen to the same topic, the message will only be sent to the browser once. Additionally, it keeps the number of open connections to the same server low. Browsers are allowed and even encouraged to impose a limit on the amount of connections to the same server, so sharing connections can be crucial. As effect, robots and maps should not open new connections whenever they need one. Instead they need to be given an existing open connection to re-use.

It might be tempting to create a single connection, give it to a `Map` object and never worry about connections again by having all robots and modules use that connection. However, that limits a map to displaying information from a single ROS system, and it is quite likely that multiple robots will all run their own ROS system. After all, if a robot can't reach the ROS master, all communication will grind to a halt and the robot will (hopefully) stop doing anything. So to ensure that the important nodes of a robot can always communicate – regardless of wireless reception – each robot needs to run its own ROS system.

That also means that each `Robot` object should have its own ROS connection. The robot objects in WebMap go even further than that and can use a different connections for different topics. That allows robots to use information from more than one ROS system, which could be useful when there is one system acting as global information server. To enable the use of multiple ROS systems, robots don't store one connection to use for publishing and subscribing. Instead they store a combination of a connection and a topic name for every topic they intend to use. These topics can be set using methods like `robot.setOdometryTopic(connection, "robot0/odom");`. When a map or robot module needs to use a topic, it should simply ask for the connection and topic name in its constructor. The same technique can be used for services, although none are used by the library yet.

2.2.2 Multiple subscriptions

Unfortunately, the rosbridge JavaScript library is a bit less convenient to use compared to the C++ library when it comes to subscribing to the same topic multiple times. With the C++ library each subscription is represented by a different object, and only when every subscription object has gone out of scope is the subscription actually terminated. The JavaScript API does support subscribing to the same topic multiple times, but unless you take extra precautions, the only way to unsubscribe from a topic removes all subscriptions to the topic.

This is generally not what you want, so to work around this rosbridge allows you to specify an identifier with every subscribe, unscubscribe, advertise or unadvertise action. When you unsubscribe with a specific

[1] `http://www.rosbridge.org`

[2] On Ubuntu, the stack can be installed by running `sudo apt-get install ros-fuerte-rosbridge-suite`

ID, only subscriptions that were made using the same ID will be removed. The same applies to advertising and unadertising topics.

In order to allow multiple robot objects or multiple modules to use the same topics, we must give every object a locally unique ID that it passes along to the rosbridge library. The ID has to be unique in context of the rosbridge connection, but it is actually easier to generate IDs that are unique within the HTML document. Those are also guaranteed to be unique for every rosbridge connection used in the document. The ID's are generated using a global counter that increments every time an ID is created. The number is prepended with the string "WebMap-" to avoid conflicts with other widgets or libraries that use the same connection.

2.2.3 Video steams

All rosbridge messages are serialized as JSON (JavaScript Object Notation), which makes them really easy to parse from JavaScript. JSON is also widely supported natively in many scripting languages or trough third party libraries in practically every programming language, which makes it a good choice for portable applications.

However, JSON doesn't natively support binary data. Such data needs to be encoded first using either Javascripts string escape sequences, or encoding schemes such as base64. The first option would inscrease the avarage data size by 105%, while base64 increases the average size by 33%.

Generally, the only binary data generated by ROS systems are images or video streams. If an image only needs to be downloaded once, there is no problem. It may take a few seconds, but that is acceptable for initialization of the interface.

Video streams are a different matter. The video streams are a lot of images sent in sequence. With a reasonably low resolution of 640 by 480 pixels, a 24 bit color depth at a rate of only 10 frames per second without any compression, that adds up to 9.216 MB/s of binary data. Using the the image_transport package[3] to compress the images can yield an average compressions rate of 0.1 by using JPEG compression.[4] The end result is still a data stream of 921.6 KB/s. After applying base64 encoding that becomes an average of about 1.226 MB/s, and that doesn't include the rosbridge and WebSocket message framing yet. At higher framerates or resolutions, the resulting datastream is unacceptably large.

A better option is to use a different method for video streaming, such as the mjpeg_server package[5]. It still uses JPEG compression on individual images, but it doesn't required base64 encoding or the WebSocket and rosbridge message framing. So instead of requiring the 1.225 MB/s as described above, it only needs 921.6 KB/s. That is an improvement, but still a significant bandwidth requirement, especially when there are multiple clients viewing the video stream at the same time.

To truly solve the problem additional measures are needed to relieve the bandwith stress on the robot. One method would be to set up a single compressed video stream to a remote server, and have that server stream the video to all viewers. That moves the bandwidth requirement from the robot to the remote server, which can use a high capacity wired connection, while nearly all robots will have a wireless connection. Using a proper video encoding such as MPEG or Theora can further reduce the required bandwidth drastically, although at the cost of CPU time and increased memory usage.

When a video stream is expected to be viewed by multiple clients at the same time, a dedicated proxy server is recommended. When the stream is expected to be viewed by only a single client at a time, the mjpeg_server package is a much simpler solution while providing adequate performance.

[3] `http://www.ros.org/wiki/image_transport`
[4] Theora compression would give a better compression rate, but it is not an option because it can't be decoded from JavaScript.
[5] `http://www.ros.org/wiki/mjpeg_server`

Chapter 3

Using the library

To use the WebMap library we need at least two thing: an (X)HTML document to add the map to and a script to do so. This chapter describes how to write both as simple as possible, and then shows a slightly more complex but also more convenient way to create robots.

3.1 Simple tutortial

In the simplest case an application consists of one (X)HTML document with one custom script. This tutorial will keep things as simple as possible, although some techniques used here may be a inconvenient for larger systems with many robots. See the next tutorial for techniques better suited for such systems.

3.1.1 Running the rosbridge server

Before anything will work, the rosbridge_server has to be running on the robot. If it is properly installed, it can be run with the following command: `rosrun rosbridge_server rosbridge.py`. On Ubuntu, the package can be installed by running `sudo apt-get install ros-fuerte-rosbridge-suite`.

3.1.2 HTML document

The first thing to do in any case is to create a simple (X)HTML document with a container for the map. It needs to include the proper scripts in the correct order. The interface uses the 'ros.js' file from the rosbridge suite, so that is included first. Secondly, the 'webmap/map.js' and 'webmap/robot.js' rely on the file 'webmap/base.js', and the 'webmap/modules.js' file relies on both 'webmap/map.js' and 'webmap/robot.js'. As last, we add a script named 'application.js' that creates the ROS connections, map and robots we want to add. The finishing touch for the head section is an included CSS stylesheet to style the document, and more importantly the map and robots.

The next thing to add is a container element for the map. A simple document that contains all required elements is included below.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="application/xhtml+xml;charset=utf-8" />
```

```
  <title>WebMap</title>

  <script type="text/javascript" src="js/ros.js"></script>
  <script type="text/javascript" src="js/webmap/base.js"></script>
  <script type="text/javascript" src="js/webmap/map.js"></script>
  <script type="text/javascript" src="js/webmap/robot.js"></script>
  <script type="text/javascript" src="js/webmap/modules.js"></script>
  <script type="text/javascript" src="js/application.js"></script>

  <link rel="stylesheet" type="text/css" href="webmap.css" />
</head>
<body>
  <h1>WebMap</h1>

  <div id="webmap_container"></div>
</body>
</html>
```

3.1.3 Application script

The next and final step is to write 'application.js'. Since we need the container element, we must wait until the DOM is done loading. We can achieve this by putting initialization code in a function and register the function as an event handler for the "DOMContentLoaded" event as demonstrated in the snippet below.

```
function init() {
  // Initialize things here.
}

document.addEventListener("DOMContentLoaded", init, false);
```

The initialization code must create the ROS connection(s), a map, and any robot we want to display. So lets start with the map:

```
function init() {
  // Get the container element.
  var container = document.getElementById("webmap_container");

  // Create a map of 700 pixels wide and 500 pixels high.
  var map = new webmap.Map(container, 700, 500);

  // Add a module to remotely control selected robots with the keyboard.
  // The parameters are linear velocity and angular velocity.
  map.addModule("Teleop", 1, 1.5);
  // Add more modules here.

  // Add the background image.
  map.addImage("background.png", -29.35, -27, 58.7, 54.0);

  // Set up a nice initial view that centers the background as large as possible.
  map.scale(500 / 58.7, map.canvas_width * 0.5, map.canvas_height * 0.5);

  // More initialization code...
}
```

The next step is to create the required ROS connections and robots. Note that attempting to use a rosbridge connection before it is ready will cause a runtime error. That's why we create the rosbridge connection first, and use its `onOpen` handler to create the robots. It is also possible to create the robot first and specify the topics to use once the connection is ready, but that would mean splitting the robot initialization code over multiple locations.

```
function init() {
  // Map initialization code...

  // Create the rosbridge connection.
  var connection = new ros.Bridge("ws://localhost:9090");

  connection.onClose = function (e) {
    // Log the message somehow, if you like.
  };

  connection.onError = function (e) {
    // Log the message somehow, if you like.
  };

  connection.onOpen = function (e) {
    // Log the message somehow, if you like.

    // Create a SVG circle to represent the robot.
    var circle = document.createElementNS(webmap.svgns, "circle");
    circle.setAttribute("cx", 0);
    circle.setAttribute("cy", 0);
    circle.setAttribute("r", 0.2);
    circle.setAttribute("draw", "black");
    circle.setAttribute("fill", "red");

    // Create the robot.
    // This doesn't actually use the rosbridge connection yet.
    var robot = new webmap.Robot("Bob", circle);

    // Set the odometry topic.
    // The rosbridge connection has to be open, or this will cause an error.
    robot.setOdometryTopic(connection, "/odom");
    robot.setTwistTopic(connection, "/cmd_vel");

    // Add a laser scanner to the robot.
    // The rosbridge connection has to be open, or this will cause an error.
    robot.addModule("LaserScanner", connection, "/base_scan");

    // Add the robot to the map.
    map.addRobot(robot);
  };
}
```

And that's it. This code should show exactly one robot with the name "Bob", represented by a red circle. Note that the name is for the users only; it doesn't need to be unique. The robot uses "/odom" as source of odometry information, and "/cmd_vel" to control the robot with twist messages. The teleop module that was added to the map earlier listens to the WASD keys of the keyboard and translates those to twist messages for the selected robot. The result can bee seen in Figure 3.1

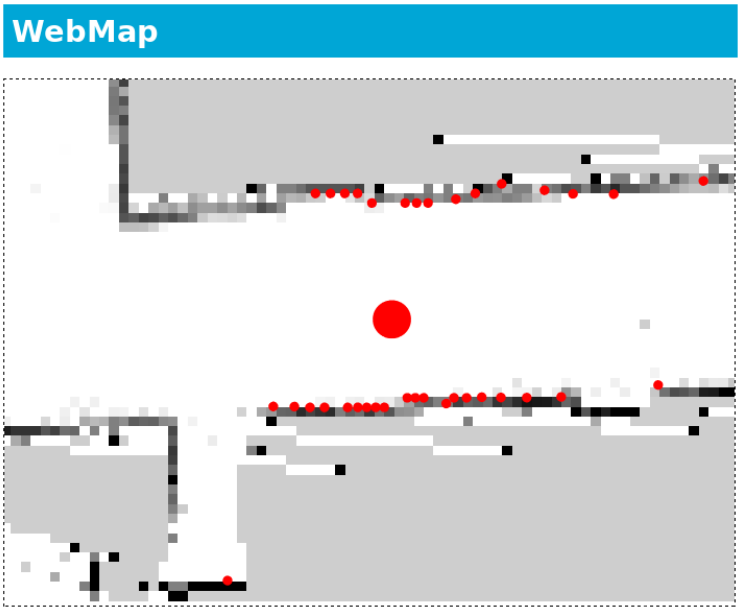


Figure 3.1: Simple tutorial screenshot.

3.2 Advanced tutorial

While the above example is fully functional, it is a bit messy. It creates SVG content directly in the event handler, and it requires us to manually call `robot.setOdometryTopic`, `robot.setTwistTopic` and `robot.addModule`. Connections and topic names could get mixed up quickly when the code is copied and pasted multiple times. Another issue is that there is no feedback when we select a robot, which is rather confusing to users. And since the robot is a circle, we have no idea which way it is facing either. This tutorial will address those issues, and immediately add logging of some messages.

3.2.1 HTML document

We will start by making all required changes to the XHTML document. We will be using another map module that displays the name of the selected robot in an HTML element. For that, we'll add a span inside a paragraph, where the span will be updated by the module. Logged messages will be added in reversed order (newest message on top) to a paragraph. So all we have to do is add two elements and a bit of text.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="application/xhtml+xml;charset=utf-8" />
  <title>WebMap</title>

  <script type="text/javascript" src="js/ros.js"></script>
  <script type="text/javascript" src="js/webmap/base.js"></script>
  <script type="text/javascript" src="js/webmap/map.js"></script>
  <script type="text/javascript" src="js/webmap/robot.js"></script>
  <script type="text/javascript" src="js/webmap/modules.js"></script>
  <script type="text/javascript" src="js/application.js"></script>
  <link rel="stylesheet" type="text/css" href="webmap.css" />
```

```
</head>
<body>
  <h1>WebMap</h1>

  <p id="log" class="log"></p>

  <p>Selected: <span id="selected"></span>.</p>
  <div id="webmap_container">
  </div>
</body>
</html>
```

3.2.2 Better SVG model

The SVG model of the previous tutorial was less than perfect. We will move the creation of the SVG content to a separate function. Then we can create a more complex SVG model without cluttering the event handler. The more complex model will show us which way the robot is facing, and with a bit of help from CSS, selected robots will be styled differently from other robots.

The new SVG model will consist of a circle and a line, grouped together. The line will extend from the center of the circle to the outer edge of the circle, in the direction the robot is facing (which is the robots positive X axis).

Of course, this is still a rather simple model. For real robots, it is recommended to create an actual top down view of the robot. It is also possible to create an SVG or PNG image of the robot in an external program and use a SVG `image` element to render it.

```
/// Create a circle model.
function circleModel(radius) {
  var svg    = document.createElementNS(webmap.svgns, "g");
  var circle = document.createElementNS(webmap.svgns, "circle");
  var line   = document.createElementNS(webmap.svgns, "line");
  svg.appendChild(circle);
  svg.appendChild(line);

  circle.setAttribute("cx", 0);
  circle.setAttribute("cy", 0);
  circle.setAttribute("r", radius);

  line.setAttribute("x1", 0);
  line.setAttribute("y1", 0);
  line.setAttribute("x2", radius);
  line.setAttribute("y2", 0);

  return svg;
}
```

3.2.3 Robot prototype

The next thing we want to get rid of is the manual calls to `robot.setOdometryTopic`, `robot.setTwistTopic` and `robot.addModule`. Manual calls to these methods can quickly become disorganized and parameters can get mixed up, especially when many robots are added to the map.

To get rid of the manual calls we will create a robot prototype. It is comparable to creating a subclass in class based object oriented languages. We will create a prototype for simulated robots from the stage robot simulator.[1] The constructor will take three arguments: a name, an index and a rosbridge connection. The name is the same as any robot name, and can be chosen freely. The index is the index number of the robot in stage, and the connection is the rosbridge connection to the system running stage.

```
/// Example robot class for Stage robots.
function StageRobot(name, index, connecion) {

  // Create the SVG content.
  var svg = circleModel(0.2);

  // Call the parent constructor, giving it the name and SVG model.
  // JavaScript does not do this automatically.
  webmap.Robot.call(this, name, svg);

  // Add CSS classes.
  this.svg.classList.add("stage");
  this.svg.classList.add("stage_" + index);

  // Set up odometry, navigation and laser scanner.
  var topic_base = "/robot_" + index;
  this.setOdometryTopic(connection, topic_base + "/odom");
  this.setTwistTopic(connection, topic_base + "/cmd_vel");
  this.addModule("LaserScanner", connection, topic_base + "/base_scan");
}

// Set up the prototype chain correctly.
webmap.extend(webmap.Robot, StageRobot);
```

There are a few things to note here. First of all, we have to call the `Robot` constructor manually since JavaScript doesn't do it for us. We can do that using the `Function.call` method, which allows us to bind any object as the `this` argument of a method.[2]

Next, we add two CSS classes to the SVG content. However, we add it to `this.svg`, not to the local `svg` variable. The reason is that the robot constructor places the SVG content inside a new group. That new group is used to set up the transformation for the robots current pose, and can also contain any SVG added by modules. By adding the CSS classes to the main robot group instead of the model alone, we ensure that we can also style the SVG of modules differently using those classes.

Finally, the call to `webmap.extend` sets up the prototype chain properly, to ensure that `StageRobot` objects are also `Robot` objects.

3.2.4 CSS

We will use CSS to style the log and the SVG content. Definitions for (X)HTML and for SVG can be mixed in the same stylesheet, but anyone is free to keep them separated in different stylesheets if they wish.

Robots add the "selected" class to their main group when they get selected. That allows us to apply different styles to a selected robot.

```
.log {
  white-space: pre-wrap;
```

[1] `http://playerstage.sourceforge.net/`
[2] `https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Function/call`

```
}

/* Prevent blurring when scaling bitmaps. */
svg.webmap {
  image-rendering:optimizeSpeed;             /* Legal fallback                  */
  image-rendering:-moz-crisp-edges;          /* Firefox                         */
  image-rendering:-o-crisp-edges;            /* Opera                           */
  image-rendering:-webkit-optimize-contrast; /* Chrome (and eventually Safari) */
  image-rendering:optimize-contrast;         /* CSS3 Proposed                   */
  -ms-interpolation-mode:nearest-neighbor;   /* IE8+                            */
}

/* Base style for stage robots. */
.stage {
  fill: #f00;
  stroke: #000;
  stroke-width: 0.01;
}

/* Selected stage robots. */
.stage.selected {
  fill: #f66;
  stroke: #fcc;
}
```

3.2.5 Init function

Finally we can use the HTML elements we added to show more information, and we can use the `StageRobot` prototype to simplify the `onOpen` handler of the rosbridge connection.

```
function init() {
  var log       = document.getElementById("log");
  var container = document.getElementById("webmap_container");

  // Create the map.
  var map = new webmap.Map(container, 700, 500);

  // Add a module to show the selected robot in the previously created span element.
  map.addModule("ShowSelected", document.getElementById("selected"));

  // Add a module to remotely control selected robots with the keyboard.
  map.addModule("Teleop", 1, 1.5);

  // Add the background image.
  map.addImage("background.png", -29.35, -27, 58.7, 54.0);

  // Set up a nice initial view that centers the background as large as possible.
  map.scale(500 / 58.7, map.canvas_width * 0.5, map.canvas_height * 0.5);

  // Create the rosbridge connection.
  var connection = new ros.Bridge("ws://localhost:9090");

  connection.onClose = function (e) {
    log.textContent = "Connection closed.\n" + log.textContent;
  };
```

```
  connection.onError = function (e) {
    log.textContent = "Error: " + e + ".\n" + log.textContent;
  };

  connection.onOpen = function (e) {
    log.textContent = "Connection established.\n" + log.textContent;
    // Add two Stage robots.
    map.addRobot(new StageRobot("Robot 0", 0, connection));
    map.addRobot(new StageRobot("Robot 1", 1, connection));
  };
}
```

Now we have a more convenient way of adding robots, with more complex SVG models that can be styled using CSS. Together with the information from the previous tutorial, this should be enough to start using the library. The result of can bee seen in Figure 3.2

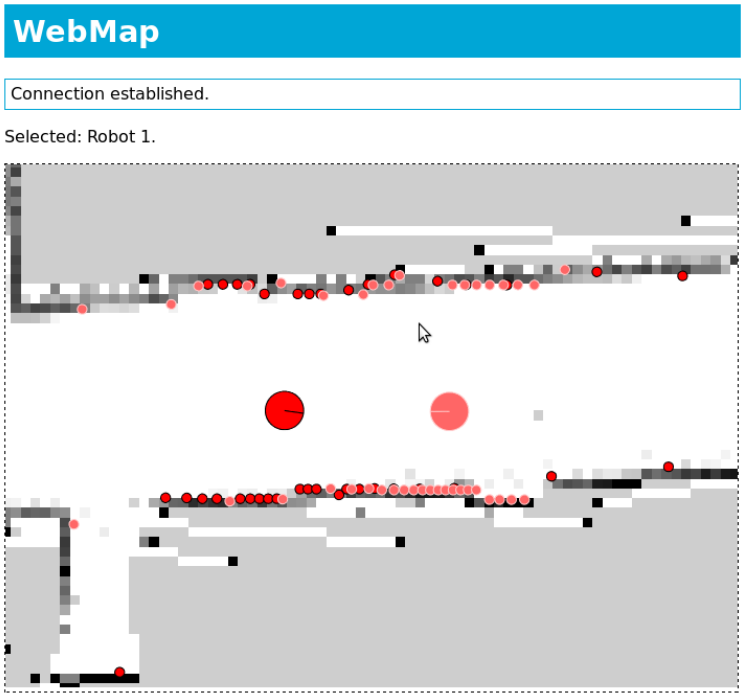


Figure 3.2: Advanced tutorial screenshot.

Chapter 4

Modules

Robots can vary wildly in their capabilities. ROS does have widely accepted methods of controlling robots and publishing sensor data, but it is impossible to forsee everything a robot can do. Additionaly, different users might want to use the interface differently. Some users might want to display additional information, or use alternate controls for panning, zooming and rotating the map. To facilitate as much use cases as possible without creating a horribly overgrown and complex library, the robot objects and the map itself can be extended using a module system.

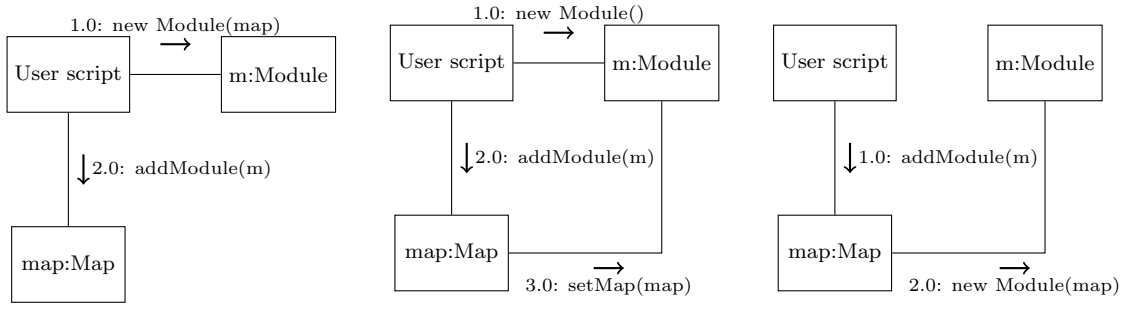
To keep the module interface consistent across robots and maps, and to prevent code duplication, both the `Robot` and `Map` objects inherit from the same `Extendable` object. Any object can be a module, there are no special requirements.

4.1 When to use modules

One thing to note is that JavaScript is already a very flexible language. Objects can be modified at runtime by adding or even removing properties and functions. Object prototypes can be modified in the same manner, since prototypes are just normal objects.

This technique can be very useful, and should be used instead of writing modules when appropriate. However, there are situations when modifying prototypes is not the best technique. A prototype is shared by all object created by the same constructor, so modifying the robot prototype will change all robots. Most robots are not identical, so there is no point in giving every single robot a laser scanner. Furthermore, since prototypes are shared across instances, only one copy of data can be stored in a prototype. That makes prototypes well suited for methods, but less suited for properties. Every property in a prototype is in essence a static class variable.

When your modifications need to remember state or shouldn't apply to all robots, a module is often more suited. This can include anything that adds SVG content and anything that reads sensor data. After all, the SVG content should be remembered somewhere so that it can be altered or removed again later, and every robot will have its own SVG content. If the goal is to only add some methods to every `Robot` or `Map` object, then modifying the prototype is the best option.



(a) Manual construction. (b) Map passed in later. (c) Automatic construction

Figure 4.1: Communication diagrams of module construction schemes.

4.2 Module instantiation

There are many different techniques that can be used to initialize modules and add them to a map or robot. The simplest method is to let the user instantiate the module manually, and then pass it to the map. This result would be code like this: `map.addModule(new webmap.modules.map.Teleop(map, 1, .5))`. While it does the job, it is a rather long line of code.

More importantly, a module often needs to know which map it will be added to, for example to register event listeners or add SVG content. That means that the module needs a reference to the map when it is being initialized. This leaves the user with the possiblity to provide a refence to one map during initialization of the module, and then actually add the module to a different map. The behvaiour of a module in such situations is undefined.

As a fix, the map can later be passed to the module by calling `module.setMap(map)` from the `map.addModule` method, instead of asking for a map in the module constructor. This removes the possibility to specify the wrong map in the module constructor, but it splits initialization code of the module over two places. Any additional parameters for the module still need to be passed to the constructor.

A third technique uses some JavaScript tricks to get rid of all issues. Instead of letting the user construct the module, the module is constructed from the `map.addModule` method. To add the teleop module with the same parameters, one would now write `map.addModule("Teleop", 1, 1.5)`. All three techniques are depicted in a communication diagram in Figure 4.1.

The first argument to the `addModule` method is a string, and must match the name of the module constructor. It is used to look up the constructor in an object acting as namespace for modules. JavaScript allows two ways to access members of an object: `object.property` and `object["property"]`. These are functionally identical, but the second method allows any arbitrary string as a member name. This makes every JavaScript object usable as associative container. The actual namespaces used are different for maps and robots; maps use `webmap.modules.map` and robots use `webmap.modules.robot`.

Once the constructor has been found it is invoked with the map as first argument, followed by any additional arguments to `map.addModule`. This is done by collecting all remaining argument to `map.addModule` in an array, with the map added as the first element. JavaScript allows to call any arbitrary method with a bound `this` argument and the normal arguments passed as array by using the `apply` method of the arbitrary method. [1] This doesn't simply work for constructors though, since they have to be invoked using special `new Constructor()` syntax.

[1] Functions and methods are also objects in JavaScript, so they can have their own methods. See `https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Function/apply` for more information on the apply method.

To work around this limitation, we define a new constructor and set its prototype to match the module prototype. The only thing the new constructor does is invoke the actual module constructor using the `apply` method, with the this argument bound to the newly created object. The constructor now creates objects that have been initialized by the module constructor and has the correct prototype, which means it is identical to an object that would be constructed using normal syntax and the same parameters. The messy details have been hidden in a separate function to keep the `addModule` method clean, and can be seen in Listing 4.1.

Listing 4.1: Construct function

```
/// Construct an object from a constructor and an array of arguments.
/**
 * \param constructor The constructor.
 * \param args        The arguments as an array.
 */
webmap.construct = function(constructor, args) {
  function F() {
    return constructor.apply(this, args);
  }
  F.prototype = constructor.prototype;
  return new F();
}
```

Another advantage of this technique is that modules are now indentified by a string, which means that robot definitions can be serialized much easier as XML or JSON. If modules need to be constructed manually, some extra glue is needed to instantiate modules based on their name. With the last method the glue is already in place. Considering all advantages, the last method is used in the library.

4.3 Reacting to events

Modules might want to react to certain events. In the case of map modules this could for example be the event that a robot was added to or removed from the map, or that the selected robot changed. Robot modules might be interested in the event that the robot received new odometry information, or a twist message. A module can listen to these events by by defining a specific method. These type of events are called triggers. Modules are not limited to responding to triggers, they can also fire them.

Firing a trigger is done by calling `robot.notifyModules("onTriggerName", [arg1, arg2, ...])`. This call will loop all modules and check for the existence of a method named `"onTriggerName"`. If it exists, it is invoked with the rest of arguments passed to `notifyModules`. The same `notifyModules` method is used internally to notify modules of triggers. See the API reference in chapter 5 for a list of available triggers, but keep in mind that any module is free to add their own.

Modules might also want to react to normal DOM events, for example to capture mouse clicks or key presses. Because of the large amount of different events, it is unfeasable to capture them all and have them delegate to `notifyModules` calls. Instead, modules that wish to react to such events can use the normal interface and register event listeners directly using `element.addEventListener`.[2] This also gives modules the option to specify exactly which SVG element to register the event listener on, which allows different actions for clicking different parts of a robot.

[2] `http://www.w3.org/TR/DOM-Level-2-Events/events.html#Events-registration`

4.4 Tutorial: Simple example

As a simple tutorial, we will write a map module that displays the name of the currently selected robot somewhere on the page. Instead of creating an element somewhere, the module will ask for an HTML element and set the text inside that element. To make the module a bit friendlier, we'll also add an optional third argument to specify a default message when there is no selected robot.

The first step is creating the constructor. Since it is a map module, it should be created in the namespace `webmap.modules.map`. And with any map module, the first argument of the constructor always has to be the map. The constructor doesn't have to do anything fancy, but it should remember the map, the container element and the default message so that they can be used later.

We will also add a deconstructor that clears the text. This is the `destroy` method that is called by the map or robot when a module is removed.

Finally, all we have to do is respond to the `onSelected` trigger, so we define a `onSelected` method. All it does is replace the text of the container element with the robot name, or the default message if no robot is selected.

The code can be found in Listing 4.2. It is now a fully functional module. To use it with a map we can write: `map.addModule("ShowSelected", document.getElementById("show_selected"))`, provided that there is an element in the document with the ID "show_selected".

Listing 4.2: ShowSelected module

```
/// Map module to show the name of the currently selected robot in any HTML element
    that can have text content.
/**
 * \param map         The map.
 * \param element     The HTML element to display the name in.
 * \param default_msg The message to show when there is no selected robot.
 */
webmap.modules.map.ShowSelected = function(map, element, default_msg) {
  // Default value for when no default_msg was specified.
  if (default_msg === undefined) default_msg = "No selection";

  // Remember the constructor arguments so we can reference them later.
  this.map                = map;
  this.default_msg        = default_msg;
  this.element            = element;

  // Call the onSelect trigger method to update the message once when the module is
      created.
  // Otherwise, the element would remain empty until the next (de)select action.
  this.onSelect();
}

/// Destroy the module and clean up.
webmap.modules.map.ShowSelected.prototype.destroy = function() {
  this.element.textContent = "";
}

/// Handle selection changes by updating the text content of the DOM element.
webmap.modules.map.ShowSelected.prototype.onSelect = function() {
  if (this.map.selected)
    this.element.textContent =  this.map.selected.name;
  } else {
    this.element.textContent =  this.default_msg;
  }
}
```

Chapter 5

API reference

This section contains a full reference of all namespaces and objects in the system. Note that unless specified, object properties are only meant for reading. They should generally not be modified directly.

5.1 WebMap namespace

Everything in the library is contained by the `webmap` namespace. Besides all object constructor, the namespace also contains some variables and functions.

5.1.1 Variables

svgns The XML namespace for SVG element.

xlinkns The XML namespace for XLink attributes.

5.1.2 Functions

`webmap.generateId()`
Generate a locally unique ID for ROS communication.

Returns a locally unique ID consisting of "WebMap-" and a number.

`webmap.extend(base, sub)`
Emulate classical inheritance by setting up a prototype chain correctly.

base The constructor of the base object type.

sub The constructor of the subtype.

`webmap.construct(constructor, args)`
Construct an object using an array of arguments for the constructor.

constructor The constructor.

args An array of arguments for the constructor.

Returns the constructed object.

5.2 Map

The `Map` object represents the main interface element and can be used to interact with the map. It has functions to zoom, rotate and translate the view of the map and to add robots. The map also provides some information on its current state, such as the currently selected robot. `Map` objects also inherit from the `Extendable` prototype to facilitate extending the map. A full overview of all methods and properties is given here.

5.2.1 Properties

Unless stated otherwise, the properties should only be read and not modified.

container Reference to the HTML element containing the map.

root Reference to the SVG root element of the map. Any overlay elements should be added to this node.

world Reference to the SVG world group of the map. Any world objects (such as robots) should be added to this node.

world_tf An SVGMatrix[1] containing the current world transformation.

selected Reference to the currently selected robot, or `null` if no robot is selected.

canvas_width The width of the canvas.

canvas_height The height of the canvas.

5.2.2 Triggers

These triggers can be used by map modules to react to certain events.

onAddRobot(robot) Fires when a robot is added to the map. The `robot` argument holds the added robot.

onRemoveRobot(robot) Fires when a robot is removed from the map. The `robot` argument holds the removed robot.

onSelect() Fires when the selection changes. The new selection can be retrieved using the `map.selected` property.

[1] `http://www.w3.org/TR/SVG/coords.html#InterfaceSVGMatrix`

5.2.3 Methods

`new webmap.Map(container, canvas_width, canvas_height)`
Create a new map.

container An HTML element to use as container for the map.

canvas_width The width of the map in pixels.

canvas_height The height of the map in pixels.

`map.getBoundingRect()`
Get the bounding rectangle of the map canvas. All coordinates are relative to the parent viewport, which will generally be the HTML document that contains the map. It can be used to translate the coordinates of mouse events.

Returns a ClientRect[2] object specifying the bounds of the map.

`map.addImage(iri, x, y, width, height)`
Create an image and add it to the world. Can be used to add a background image to represent the map.

iri The IRI[3] identifying the image to add.

x The X coordinate of the image in the world.

y The Y coordinate of the image in the world.

width The width of the image in the world.

height The height of the image in the world.

Returns a reference to the created image.

`map.removeImage(img)`
Remove an image that was previously added by a call to `map.addImage`.

img The image to remove, as returned by `map.addImage`.

`map.addRobot(robot)`
Add a robot to the map. If the robot is already on a different map, it is removed from that map first.

robot The robot to add.

`map.removeRobot(robot)`
Remove a robot from the map. Note that the robot does not unsubscribe from ROS topics automatically.

robot The robot to remove.

`map.setSelected()`
Set the currently selected robot. Passing `null` instead of a robot clears the selection.

robot The robot to select, or `null` to clear the selection.

[2] `http://www.w3.org/TR/cssom-view/#the-clientrect-interface`
[3] `https://tools.ietf.org/html/rfc3987`

`map.translate(x, y)`
Translate the view by an amount of pixels.

x The X offset to translate by, in pixels.

y The Y offset to translate by, in pixels.

`map.scale(factor, x, y)`
Scale the view around a point. The optional arguments allow you to scale around a point on the viewport. The center should be specified in screen coordinates with the origin at the top left corner of the map. The specified point will remain at the same screen coordinates after the scaling.

factor The factor to scale the world by.

x (Optional) The X coordinate of point on the viewport to scale around.

y (Optional) The Y coordinate of point on the viewport to scale around.

`map.rotate(angle, x, y)`
Rotate the view around a point. The optional arguments allow you to rotate the world around a point on the viewport. The center should be specified in screen coordinates with the origin at the top left corner of the map. The specified point will remain at the same screen coordinates after the rotation.

angle The angle to rotate the world by.

x (Optional) The X coordinate of point on the viewport to rotate around.

y (Optional) The Y coordinate of point on the viewport to rotate around.

5.3 Robot

Every robot on the map is represented by a `Robot` object. A `Robot` object maintains state information such as its current pose and SVG content that can be used to render the robot. A map can render a robot by adding the SVG content of the robot to the map. Additionally, the `Robot` objects can be used to control robots by sending twist messages or navigation goals. Like `Map` objects, `Robot` objects also inherit from the `Extendable` prototype. A full overview of all methods and properties is given here.

5.3.1 Properties

Unless stated otherwise, the properties should only be read and not modified.

name The human readable name of the robot.

selected True if the robot is selected.

map A reference to the map the robot is currently being rendered on, or null.

position A vector object with x, y and z attributes representing the position of the robot.

orientation A quaternion object with w, x, y, z attributes representing the orientation of the robot.

angle The angle of the robot on the XY plane with the X axis, in degrees. This can be more convenient than the quaternion representation.

base_link The base link of the robot, to use with the navigation stack.

valid If true, the pose information cached by the robot is considered valid.

svg A reference to the SVG group holding the visual robot representation.

5.3.2 Triggers

These triggers can be used by robot modules to react to certain events.

onSendTwist(msg) Fires when the object sends a twist message to the robot. The `msg` argument holds the message that was sent to the robot.

onSendMove(msg) Fires when the object sends a navigation goal to the robot. The `msg` argument holds the message that was sent to the robot.

onSelect(selected) Fires when the robot is (de)selected. The `selected` argument is true if the robot was selected, false otherwise.

onOdometryUpdate() Fires when the robot received new odometry information. Only fires if the odometry data is different from the cached data.

5.3.3 Methods

`new webmap.Robot(name, model)`
Create a new robot.

name The human readable name of the robot.

model An SVG element to represent the robot.

`robot.destroy()`
Stop the ROS communication and remove the robot from any map it is on.

`robot.setOdometryTopic(connection, topic, throttle)`
Set and subscribe to the odometry topic used by the robot. The topic can use one of Pose, PoseStamped, PoseWithCovarianceStamped or Pose2D messages from the geometry_msgs package.

connection The ROS connection for this topic.

topic The name of the topic.

throttle (Optional) The minimum time in milliseconds between receiving two updates. Defaults to 100.

`robot.unsubscribeOdometryTopic()`
Unsubscribe the robot from the odometry topic. This method doesn nothing if the robot wasn't subscribed to an odometry topic.

`robot.setTwistTopic(connection, topic)`
Set and advertise the twist topic used to control the robots movement. The topic must use geometry_msgs/Twist messages.

connection The ROS connection for this topic.

topic The name of the topic.

`robot.unadvertiseTwistTopic()`
Unadvertise the twist topic. This method doesn nothing if the robot hasn't advertised a twist topic.

`robot.setMoveTopic(connection, topic)`
Set and advertise the goal topic used by the robot's navigation server. The topic must accept geometry_msgs/PoseStamped messages.

connection The ROS connection for this topic.

topic The name of the topic.

`robot.unadvertiseMoveTopic()`
Unadvertise the move topic. This method doesn nothing if the robot hasn't advertised a move topic.

`robot.twist(x, y, z, rx, ry, rz)`
Send a twist message to the robot.

x The linear X component.

y The linear Y component.

z The linear Z component.

rx The angular X component.

ry The angular Y component.

rz The angular Z component.

`robot.move(position, orientation)`
Send a move goal to the navigation server of the robot.

position A 3D vector object with x, y and z attributes representing the desired position of the base link.

orientation A 4D quaternion object with w, x, y and z attribute representing the desired orientation of the base link.

5.4 Extendable

The `Extendable` object provides a generic interface for objects that can be extended with modules. It is not genereally instantiated directly. Instead, objects can inherit from the Extendable object to use the module system. Both the `Map` and `Robot` objects inherit from this.

5.4.1 Methods

`new webmap.Extendable(namespace)`
Create a new extendable object. This is generally only invoked from objects inheriting from Extendable.

namespace The namespace to search when adding modules.

`extendable.addModule(name, ...)`
Add a module. Pass in any additional module arguments after the name argument. The modules constructor will be called with the map as the first argument, followed by any additional arguments you specify here.

name The name of the modules.

... The remaining arguments for the module constructor.

Returns a reference to the created module, which can be passed to removeModule().

`extendable.removeModule(module)`
Remove a module.

module The module to remove, as returned by the addModule method.

`extendable.notifyModules(handler, ...)`
Notify all modules by calling a member function. Additional arguments will be passed to the invoked method. If a module doesn't have the method, nothing happens for that module.

handler The name of the method to invoke.

... The remaning arguments will be passed to the invoked method.

5.5 Map modules

The library includes three map modules to add additional functionality to the map.

`map.addModule("MouseViewControl")`
Change the view of the map with mouse controls. Drag the map to translate, use the scroll wheel to zoom, or scroll with shift pressed to rotate the map.

`map.addModule("ShowSelected", element, default_msg)`
Shows the name of the selected robot in an HTML element. The entire content of the element will be replaced by the name of the robot.

element The HTML element to display the name in.

default_msg (Optional) The message to show when there is no selected robot.

`map.addModule("Teleop", speed, rotation_speed)`
Control the selected robot using the WASD keys.

speed The forward speed of the robot.

rotation_speed The rotation speed of the robot, in rad/s.

5.6 Robot modules

`robot.addModule("LaserScanner", connection, topic, x, y, angle, max_dots, throttle)`
Visualize the readings of a laser scanner. The module can only read sensor_msgs/LaserScan messages.

connection The ROS connection to use.

topic The name of the topic to subscribe to.

x The X offset of the sensor from the origin of the robot.

y The Y offset of the sensor from the origin of the robot.

angle The rotation of the sensor in degrees relative to the robot, 0 being aligned with the X axis.

max_dots (Optional) The maximum number of dots to render. Defaults to 50.

throttle (Optional) The minimum time in milliseconds between ROS messages. Defaults to 300.

Appendix A

Background knowledge

In order to read the report, some knowledge is required on part of the reader regarding the Document Object Model and JavaScript. This chapter will give a short explanation of both.

A.1 Document Object Model

The Document Object Model (DOM) is an object oriented representation of HTML and XML documents. A document is represented as a tree, where every HTML or XML element is a node in that tree. There is always exactly one root node, also called the document node. In the case of HTML documents, that is the `<html>` element. Nodes can also be other things than XML or HTML elements, such as the attributes of an element or the text nodes between elements.

The DOM also describes a set of methods that can be performed on nodes, such as adding, removing or swapping child nodes. By manipulating the DOM tree, the document can be changed on the fly to provide a dynamic interface for the user.

A.2 JavaScript

JavaScript is an object oriented prototype based scripting language. The most important difference with traditional class based languages is that JavaScript has no notion of classes.

Prototypes Instead of classes, objects can have a prototype. This prototype is an implicit property of the object. The link to the prototype exists, but the programmer can not see or alter the link.

When accessing object members, the object is searched for the requested member first. If that fails, and the object has a prototype, the prototype is searched next. Since a prototype is nothing more than an object, the prototype itself can also have a prototype. If it has one, that prototype is searched next. This continues until the member is found or there are no prototypes left. This prototype chain is somewhat comparable to a class hierarchy.

Functions One thing to know is that in JavaScript, all user-created methods and functions are also objects. Like most objects, functions also have a prototype. This prototype defines a set of useful methods to bind

function arguments, or to call a function with an array of arguments instead of a normal argument list. And because functions are objects, you can assign functions to member variables of an object at any time.

Object construction Objects are constructed using the "`new`" keyword: `new Foo(arg1, arg2, ...)`. Here, Foo is a function that initializes an object. Within the constructor, the variable `this` refers to the object being constructed. Whenever an object is constructed, the "`prototype`" property of the constructor is copied and becomes the prototype of the constructed object. In this case that means that `Foo.prototype` becomes the prototype of objects created by `new Foo()`. However, `Foo.prototype` is not the prototype of `Foo`. For `Foo` it is just a regular property that happens to be named "prototype".

Prototype modification It is possible to change `Foo.prototype` by adding or removing methods and properties, or you can replace it entirely. Note however if you create an object first, and then *replace* `Foo.prototype`, the old object is not affected. However, if you *add* a method to `Foo.prototype`, the old object is affected because the objects prototype is merely a reference to whatever `Foo.prototype` was when the object was created. This is demonstrated in the example below.

```
function Foo() {
  // Initialize object.
  // Don't need to do anything now.
}

// Add a method to the prototype.
Foo.prototype.showMsg = function() {
  alert("Foo");
}

var a = new Foo();
a.showMsg(); // Says "Foo".


// Overwrite the previously created method.
Foo.prototype.showMsg = function() {
  alert("Bar");
}
a.showMsg(); // Says "Bar".

// Replace the entire prototype.
Foo.prototype = new Object();
Foo.prototype.showMsg = function() {
  alert("New prototype!");
}

var b = new Foo();
a.showMsg(); // Still says "Bar".
b.showMsg(); // Says "New prototype!".
```

Prototype based inheritance A common application of replacing the prototype property of a constructor is to set up prototype chains. This is analogous to inheriting from a class in traditional class based languages. A key difference though is that you inherit from an instance, and not a class. The instance has to be constructed first and must be given the proper arguments for its constructor. The example below demonstrates this technique to make methods of `Foo` available in `Bar`.

```
function Foo() {
  this.msg = "Foo";
}

Foo.prototype.showMsg = function() {
  alert(this.msg);
}

function Bar() {
  this.msg = "Bar";
}

Bar.prototype = new Foo();

var a = new Foo();
var b = new Bar();

a.showMsg(); // Shows "Foo";
b.showMsg(); // Shows "Bar";
```

Note here that `Bar.prototype` is an instance of `Foo`. And since it is an instance of Foo, it also has a prototype, namely `Foo.prototype`. So modifying `Foo.prototype` by adding or removing members will affect both object `a` and object `b`, but modifying `Bar.prototype` will only affect object `b`. If we instead write `Bar.prototype = Foo.prototype`, then modifying `Bar.prototype` would also affect object `a`, but this is generally not desired.