FORD OTOSAN AUTONOMOUS TRUCK HMI SOFTWARE DOCUMENTATION

Ali Ekin Gurgen

August 2020

Table of Contents

Introduction

The purpose of this document is to provide implementation details and documentation for the preliminary work conducted on Ford Otosan's Human-Machine Interface project for autonomous trucks. Majority of the current work on this project has been carried out by Ali Ekin Gurgen during his internship between June - August 2020, under the supervision of Sertac Akin and Berzah Ozan. The goal of the project was to *"design and implement a web-based human-machine interface application that would ultimately run on an Android tablet's web browser next to the driver and visualize the autonomous driving data in real-time."*

We have used the open-source Uber Autonomous Visualization System (AVS) toolkit for the backend and frontend of this system, primarily due to 3 main advantages that it had compared to other frameworks:

1) Providing a web-based system that enables our application to be platform independent
2) Providing visualization tools and abstractions that are specific to autonomous driving
3) Being an open-source toolkit which makes not only free-to-use but also very flexible

However, using Uber AVS also caused certain disadvantages, such as:

1) Being a relatively new platform, Uber AVS had a limited amount of online resources and certain unsupported features, especially for Special Topics (discussed at the end of this document)
2) Being a software directed specifically towards autonomous driving, it has certain uncustomizable features, such as the necessity to know a "vehicle_pose" data stream at all times

Uber AVS consists of two main components: XVIZ and streetscape.gl. XVIZ is a protocol for real-time transfer and visualization of autonomy data, and streetscape.gl is a React and WebGL based frontend framework for visualizing robotics data. Both of

these components consist of JavaScript libraries, and a fundamental knowledge of JavaScript, React and 2D and 3D graphics rendering are needed to work on this project.

The preliminary work conducted on this project involved working with an example rosbag data gathered from running an autonomous driving scenario on the Town 4 map on the Carla SImulator. Last section, Special Topic 2: Working with Live Data, provides more details about integrating this HMI system with a live ROS system, whereas the rest of the document assumes the static loading of data from a rosbag file. Code that is discussed in this document is available in the ford-autonomous-vehicles-hmi repository.

This document will explain the implementation details in three main sections: The first section will discuss the top level communications between backend and the frontend. The second section will explain the files composing the backend in more detail, which has the primary goal of converting the autonomy data from ROS into the XVIZ protocol. The third section explains the streetscape.gl-based frontend components in more detail. And finally, we will give specific attention to certain special topics such as importing HD Map data into our application and moving from our static-loading example to a live stream of ROS data.

Section 1: Overview of Top Level Components

Before going into details of our application's implementation, we will first define the technologies used. Table 1, given below, provides a short description of each technology we have used.

Table 1: Brief description of technologies used in the implementation of our application

XVIZ	Data abstraction protocol that has been developed for Uber AVS. We convert ROS data to XVIZ.
JavaScript	Primary language used for both the backend and the frontend.
React	Open-source JavaScript frontend framework. streetscape.gl is entirely composed of React components.
WebGL	JavaScript API for rendering interactive 2D and 3D graphics in web browsers. We used the WebGL-powered framework deck.gl.

HTML & CSS	HTML is the Standard markup language for displaying content on the web. Cascading Style Sheets (CSS) are used for styling UI components.
Web Browser	Primarily used Firefox and Google Chrome during development.

Figure 1, given below, shows the data flow from its starting point in the rosbag file to its destination point in the web browser. Notice that this flow assumes a static loading of the ROS data. How this data flow changes when data comes from a live ROS system is discussed in Special Topic 2: Working with a Live ROS System.

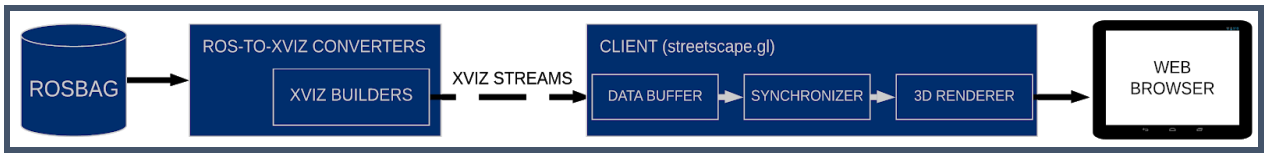

Figure 1: Data Flow from the rosbag file to web browser

As mentioned earlier, the backend of our application uses the JavaScript libraries for XVIZ, specifically its @xviz/ros API. Primary functionality of this component is to convert relevant data from ROS into XVIZ streams. Code written for this component is available under the "ros-to-xviz" directory of our GitHub repository. Frontend of our application uses the JavaScript libraries provided with the streetscape.gl framework. This section makes use of React as well as the WebGL-powered visualization framework called deck.gl. This component's primary functionality is to visualize the XVIZ streams in a web browser, and the code we have written is available under the "react-app" directory. The specific instructions for building and running these two components are given below:

To Build & Run:

ros-to-xviz:

- **Build:** *yarn bootstrap*
- **Run:** *node index.js server -d [path/to/rosbag/] --rosConfig config-test-custom.json*

react-app:

- **Build:** *yarn*

- **Run:** *yarn start-streaming*

Browser:

- *localhost:8080/[name_of_rosbag]*

An important point to note is that even though our application is web-based, all of the data needed is available locally (whether it is a static rosbag file saved on disk, or a live ROS system running on the self-driving car), and all of the communications are carried out on the local network. Ubuntu 18.04 was the primary operating system used during the development of this project and the end product of this application is expected to run on a LInux-based computer on the autonomous vehicle.

Section 2: Backend

As mentioned above, the backend of the application has the primary goal of converting the statically loaded ROS autonomy data into the XVIZ protocol. To achieve this, we have used the @xviz/ros API, and specifically the ROS-to-XVIZ converters (called ROS2XVIZConverter) that it provides. Loading and converting ROS data is carried out in 3 main steps:

1) Define relevant ROS topics and corresponding XVIZ streams in a config file
2) Read the raw data from the rosbag and perform the necessary conversions
3) Use XVIZBuilder and XVIZMetaBuilder to build the XVIZ streams using the data

We will now discuss each of these steps in more detail by using the fundamental "vehicle_pose" data stream as an example.

2.1: Creating the Configuration File

Configuration file (called "config-test-custom.json") is a JSON file that is structured in a specific way to define each relevant ROS topic, its type, and the corresponding XVIZ stream and converter. For example, "/carla/ego_vehicle/odometry" topic from the rosbag, which contains the fundamental position and orientation data of our vehicle and has type "nav_msgs/Odometry", is converted to the "/vehicle_pose" XVIZ stream using the "Sensor Odometry" converter. The information given in the preceding sentence is defined in the configuration file as shown in Figure 2:

```
{
  "topic": "/carla/ego_vehicle/odometry",
  "type": "nav_msgs/Odometry",
  "converter": "SensorOdometry",
  "config": {
    "xvizStream": "/vehicle_pose"
  }
},
```

Figure 2: Defining the configuration details of ros-to-xviz conversion

The next step of performing the ROS-to-XVIZ conversion is to define the relevant ROS2XVIZConverter. In this case this, we have defined the "SensorOdometry" converter in the "odometry-converter.js" file, which is under the messages directory. This converter performs our conversion process' following two steps, which are discussed below.

2.2: Reading and Modifying the ROS Data

First step performed in ROS2XVIZConverter is to use JSON unpacking to read the relevant information from our ROS topic. Figure 3, given below, how this is done for reading the x, y, z positions that are contained in the "/carla/ego_vehicle/odometry" ROS topic. Similarly, Figure 3 shows how we read the orientation data from the ROS topic, which is the other important data to read from this ROS topic to be able to construct the "vehicle_pose."

```
// Read Position
var {
  pose: {
    pose: {
      position: {x, y, z}
    }
  }
} = message;
const pos = [x, y, z];
```

Figure 2: Unpacking the position data from the ROS topic

```
// Read Orientation
var {
  pose: {
    pose: {
      orientation: {x, y, z, w}
    }
  }
} = message;
```

Figure 3: Unpacking the orientation data from the ROS topic

Before building the "vehicle_pose" XVIZ stream, we need to perform any necessary modifications and conversions on the data we have just loaded. In this case, orientation data given in the ROS topic is in the x-y-z-w quaternion format, while the XVIZ protocol uses the Roll-Pitch-Yaw convention. Figure 4, given below, shows how this conversion is performed.

```
// Convert orientation (from Quaternions to Roll-Pitch-Yaw)
const roll = Math.atan2(2*x*w + 2*y*z, 1 - 2*x*x - 2*y*y);
const pitch =  Math.asin(2*w*y + 2*z*x);
const yaw = Math.atan2(2*z*w + 2*x*y, 1 - 2*y*y - 2*z*z);
```

Figure 4: Converinting loaded data from Quaternions to Roll-Pitch-Yaw

2.3: Building the XVIZ Stream

The final step of the ROS-to-XVIZ conversion is to build the XVIZ streams from the data we have just read and processed. Fortunately, @xviz/ros modules provide two useful tools for this task: XVIZBuilder and XVIZMetaBuilder. Figure 5, given below, shows how these tools are used to build the "vehicle_pose" XVIZ stream.

Notice that, in this example, ".mapOrigin" attribute of xvizBuilder has a hard-coded longitude, latitude and altitude values. These values define the origin of the Carla map used in this example and everything other position in the visualization is defined by using offsets from this map origin. For example, ".position" defines the autonomous vehicle's position offset from the map origin in the x-y-z format.

```
    xvizBuilder
      .pose('/vehicle_pose')
      .timestamp(TimeUtil.toDate(timestamp).getTime() / 1e3)
      .mapOrigin(29.008258, 41.045199, 40)
      .orientation(roll, pitch, yaw)
      .position(pos[0], pos[1], pos[2]);
  }

  getMetadata(xvizMetaBuilder) {
    xvizMetaBuilder
      .stream('/vehicle_pose')
      .category('pose');
  }
}
```

Figure 5: Building the "vehicle_pose" XVIZ stream

The 3-step procedure described above is also performed for the other important ROS topics that need to be visualized in the frontend. Some of the other converters that are defined under the messages directory of "ros-to-xviz" are: vehicle-status-converter.js, behavior-state-converter.js, turn-signal-converter.js, waypoints-converter.js, etc.

The XVIZ streams that are built using the described 3-step procedure are then streamed locally using port 3000.Our frontend application communicated with the backend and received the XVIZ streams by using this port number. Next, we will go into the implementation details of the frontend application and give an example of how the XVIZ stream becomes a visual element on the web browser.

Section 3: Frontend

As mentioned earlier, the code written for the front end of our application is available under the "react-app" directory. The .obj files and images used are available under "assets," and all of the source code is under "src" directory. "app.js" file contains main React component and it contains several other important React components such as the ControlPanel (panel shown on the left side of the application screen), MapView (component that does 3D rendering of the map, vehicle, and surrounding objects), PlaybackControl (playback control panel at the bottom of the screen), and finally

TurnSignalHud (hud at the top right corner of the screen showing turn signals). Visualizing the XVIZ data is performed in a 2-step procedure:

1) Use streetscape.gl's file loader to load XVIZ data
2) Import the relevant React component and put the data into it

3.1: Loading XVIZ Data

Fortunately, streetscape.gl's API provides useful tools for loading the XVIZ streams for both statical or in a live loading scenarios. In this case, our application uses the static loading of the XVIZ data. FIgure 6, given below, shows how we use the XVIZLoaderFactory to perform the loading operation. Notice that the buildLoaderOptions() function, which is defined in app.js, provides the important configuration details of how to perform the loading of our XVIZ data.

```
// __IS_STREAMING__ and __IS_LIVE__ are defined in webpack.config.js
const exampleLog = XVIZLoaderFactory.load(__IS_STREAMING__, __IS_LIVE__, buildLoaderOptions());
```

Figure 6: Loading the XVIZ Data Statically

3.2: Creating the React Components for Visualization

In a similar fashion to the previous section, we will go over an example to show how the XVIZ data is visualized on the screen. THe example we will consider for this section is the "SpeedInfo" which visualizes the "vehicle/velocity" and "vehicle/acceleration" XVIZ streams, and it is a sub-component of the ControlPanel component that we have mentioned earlier. After importing the relevant React components from the streetscape.gl modules, the code given below is the "render()" method of the "SpeedInfo" component. As you will notice, "SpeedInfo" component simply returns two instances of the "MeterWidget" component that is imported from the default modules of streetscape.gl (one for velocity and another acceleration) as well as an instance of a "BaseWidget," that is used for rendering the traffic sign image.

```
render() {
    return (
        <div style={{display: 'flex', flexDirection: 'row', justifyContent:'space-between', alignItems:'center'
            <MeterWidget
            log={this.props.log}
            style={METER_WIDGET_STYLE}
            streamName="/vehicle/velocity"
            units="Speed"
            min={0}
            max={20}
            />
            <MeterWidget
            log={this.props.log}
            style={METER_WIDGET_STYLE}
            streamName="/vehicle/acceleration"
            units="Acceleration"
            min={-4}
            max={4}
            />
            <BaseWidget log={this.props.log} streamNames={{state: '/vehicle/speed_limit'}}>
                {this._renderSpeedLimit}
            </BaseWidget>
        </div>
    );
}
```

Figure 7: render() Method of the SpeedInfo Component

Using the imported MeterWidget for visualization is pretty straightforward and one can see streetscape.gl's documentation to get further insights about how to use it. The "BaseWidget" we used for rendering the traffic sign visualization, on the other hand, is a good example of creating custom React Components. Figure 8, given below, shows the "_renderSpeedLimit()" method which currently uses a hard-coded "limit" value to add the relevant speed limit image from assets as a simple HTML image. Ideally, if our ROS data would contain a speed limit data, and the "limit" would be read from that image.

```
_renderSpeedLimit({streams}) {
    var limit = 110;
    return (
      <div>
        {<img src={"../assets/speed-limit-"+ limit +".png"} alt="Speed-Limit-Image" className="speed-limit-image" />}
      </div>
    );
}
```

Figure 8: _renderSpeedLimit() Method for Adding Speed Limit Image to SpeedInfo

A good example of using "real" data for creating such custom components is the steering wheel image that is rendered under the SteeringInfo component, which is also

a sub-component of the ControlPanel component. Figure 9, given below, shows how the "vehicle/wheel_angle" XVIZ stream is used to transform the wheel image so that the wheel image displayed on the screen is being rotated in real time as the autonomous vehicle's orientation changes.

```
_renderSteeringWheel({streams}) {
    var angle = (streams.state.data && streams.state.data.variable) || 'unknown';
    var rot = {
      transform: `rotate(${angle}deg)`
    };
    return (
      <div style={rot}>
        {<img src={wheelImageLocation} alt="Wheel" className="wheel-image" />}
      </div>
    );
}
```

Figure 9: _renderSteeringWheel() Method for Adding Steering Image to SteeringInfo

The advantage of using React components in our application is that when the data in the defined XVIZ streams change with time, the corresponding React components automatically get updated, allowing us to have a real-time visualization of the data in our application.

Special Topic 1: Importing HD Map Data

Our frontend framework streetscape.gl provides certain functionalities to render custom map layers. Our team is planning to use a custom HD map in the OpenDrive format. Therefore our task is to statically load this HD map and visualize it in the HMI application. This task can be accomplished in three main steps:

1) Convert OpenDrive format to Lanelet and save the resulting XML file
2) When the application runs, read and parse the XML file to extract the relevant information such as roads, lanes, traffic lights, traffic signs, etc.
3) Use streetscape.gl's custom 3D layers to visualize the HD map

We will now explain these three steps in further detail.

Step 1: OpenDrive to Lanelet (XML)

Carla Simulator's "Town 4" map was used during the development of this example application. This map can be downloaded here in the OpenDrive format. We used the open-source *opendrive2lanelet* tool to convert OpenDrive format to Lanelets in XML format. You can read the documentation of this tool and take a look at some examples to gain further insight about the installation process and usage of the tool. Figure 10, given below, shows small portions of the HD map data in OpenDrive format (before conversion) and XML format (after conversion).

```
<road name="Road 0" length="3.9509958168569284e+1" id="0" junction="-1">
    <link>
        <predecessor elementType="junction" elementId="106"/>
        <successor elementType="junction" elementId="281"/>
    </link>
    <type s="0.0000000000000000e+0" type="town">
        <speed max="55" unit="mph"/>
    </type>
    <planView>
        <geometry s="0.0000000000000000e+0" x="2.1132980814927132e+2" y="3.0962988714087493e+2"
            <line/>
        </geometry>
    </planView>
    <elevationProfile>
        <elevation s="0.0000000000000000e+0" a="4.3487548828125000e-3" b="0.0000000000000000e+0"
    </elevationProfile>
    <lanes>
```

```
<lanelet id="100">
  <leftBound>
    <point>
      <x>211.32981</x>
      <y>309.62989</y>
    </point>
    <point>
      <x>211.82991</x>
      <y>309.62474</y>
    </point>
    <point>
      <x>212.33001</x>
      <y>309.61959</y>
    </point>
```

Figure 10: HD Map Data in OpenDrive Format (left) and XML Lanelet format (right)

Step 2: Read and Parse the XML File

The important point to note is that, we have performed Step 1 independently of the HMI application, and now in runtime of the application's frontend, we need to read and parse the HD map XML file which is saved on the local disk. This step is performed within the *getxml()* function inside the *map-view.js* file. Since the current Town 4 map only contains road data, we are only converted with parsing the left and right bounds of the Lanelet lanes and storing them in a global variable, which is later used in the visualization task in Step 3. This function is called in the beginning of the file, allowing the map data to be gathered before the application's loading is completed. Function's code is also given below in Figure 11. This function's specific details might change depending on the structure of the HD map's XML file but the general idea will stay the same.

```
function getxml() {
  const xml_string = require('./output_file.xml');

  const xml2js = require('xml2js');
  const parser = new xml2js.Parser({ attrkey: "ATTR" });

  parser.parseString(xml_string, function(error, result) {
      if(error === null) {
          for (var i = 0; i < 300; i++) {
            var path = [];
            for (var j = 0; j < result.commonRoad.lanelet[i].leftBound[0].point.length; j++) {
              path.push([parseFloat(result.commonRoad.lanelet[i].leftBound[0].point[j].x[0]),
                      parseFloat(result.commonRoad.lanelet[i].leftBound[0].point[j].y[0]),
                      0])
            }
            myData.push({path});
          }
          for (var i = 0; i < 300; i++) {
            var path = [];
            for (var j = 0; j < result.commonRoad.lanelet[i].rightBound[0].point.length; j++) {
              path.push([parseFloat(result.commonRoad.lanelet[i].rightBound[0].point[j].x[0]),
                      parseFloat(result.commonRoad.lanelet[i].rightBound[0].point[j].y[0]),
                      0])
            }
            myData.push({path});
          }
      }
      else {
          console.log(error);
      }
  });
}
```

Figure 11: getxml() function which parses the XML file containing the HD map data

Step 3: Visualization of the HD Map Information

HD map data that has been stored in a global variable (myData object in Figure 11) is being visualized on the map view of the application through streetscape.gl's custom 3D layers. These layers are inherited from the underlying deck.gl layers. Specifically, for the purpose of visualizing the HD map roads, we need to use the LaneLayer for rendering the HD map data, which is then added as a custom layer to the main

LogViewer component. Code given below shows how this custom LaneLayer is defined.

```
new LaneLayer({
  id: 'lanes',
  coordinate: COORDINATE_SYSTEM.METER_OFFSETS,
  coordinateOrigin: [8, 49],
  //coordinateOrigin: [29.0081, 41.0465],

  data: myData,

  getPath: d => {console.log(d.path); return d.path;},
  getColor: [80, 200, 0],
  getColor2: [0, 128, 255],
  getWidth: [0.1, 0.05, 0.05],
  getDashArray: [4, 1, 1, 1]
}),
```

Figure 12: Custom LaneLayer used for visualizing HD map road data

Similarly, other provided layers such as SignLayer and TrafficLightLayer can be used to visualize the traffic signs and lights. Even though the Town 4 map does not contain such map features, examples of these layers are added to the application for demonstration purposes. In a similar manner, one can use these provided custom 3D layers as an example to define completely new 3D layers for rendering of new HD map features.

Special Topic 2: Working with a Live ROS System

So far, we have been considering the static loading case for our HMI application. We will now briefly touch upon the real time visualization of the ROS data using Uber AVS. Although our current application does not work for the live visualization case, some preliminary work has been done about it. Visualization of a live stream of ROS data will work in three main steps:

1) Have a live ROS application running.
2) Write JavaScript code that will connect to the live ROS application to extract relevant topics and convert them to JS objects in real time.
3) Convert JS objects to the XVIZ in real time, using XVIZBuilder and XVIZMetaBuilder.

4) Use XVIZ's "i/o" and "server" modules to create a websocket connection between the backend and the frontend and transfer the XVIZ data to the frontend to visualize in real time.

Step 1: Live ROS Application

Ideally this would be a real autonomous vehicle's ROS system, but in our sample test case, we can use the rosbag play command line tool to play our rosbag as a live ROS application.

Step 2: Connect to ROS and Convert Data to JS in Real Time

We use the rosnodejs package to connect to the running ROS application, pull the relevant ROS topics and convert them to JS objects. Code given below shows how this connection and conversion is performed for the vehicle's position and orientation data.

```
const rosnodejs = require('rosnodejs');
rosnodejs.initNode('/my_node')
.then(() => {
  console.log("Waiting for data...")

});

const nh = rosnodejs.nh;
const sub = nh.subscribe('/carla/ego_vehicle/odometry', 'nav_msgs/Odometry', (msg) => {

  var {
    header: {
      stamp: timestamp
    }
  } = msg;

  var {
    pose: {
      pose: {
        position: {x, y, z}
      }
    }
  } = msg;
  const pos = [x, y, z];
```

Figure 13: Connection to ROS and pulling the relevant data from the ROS topic

Step 3: Convert JS Objects to XVIZ Protocol in Real Time

XVIZ's standard XVIZBuilder and XVIZMetaBuilder can be used to convert the JS objects into the XVIZ protocol. Code given below shows how the position and orientation data are being used to create the "vehicle_pose" XVIZ stream.

```
xvizBuilder
  .pose('/vehicle_pose')
  .timestamp(TimeUtil.toDate(timestamp).getTime() / 1e3)
  .mapOrigin(8, 49, 0)
  .orientation(roll, pitch, yaw)
  .position(pos[0], pos[1], pos[2]);
xvizMetaBuilder
  .stream('/vehicle_pose')
  .category('pose');
```

Figure 14: Conversion of JS objects to XVIZ

Step 4: Create a Real Time Backend-Frontend Connection

This is the most important step of creating a HMI application that can visualize data in real time. Unfortunately, during my internship duration, I did not have a chance to accomplish this task, but I suggest taking a look at documentation for XVIZ's io and server modules as well as the issues section of the XVIZ GitHub repository.