FAQ »

Configuration and Operation

How can I change the frame rate that is used for calling a plugins' *update()* method?

The internal frame rate of the ModuleManager can only be set for all plugins at once. If plugins shall run at different rates, this must be handled within the *update()* methods of the plugins and the rates of all plugins must, of course, be integer fractions of the base rate. An alternative method may also to run multiple instances of the ModuleManager, each with a frame rate which is convenient to its own plug-ins individual frame rates.

In order to change the base frame rate of the moduleManager, use the command line argument *-t <frameTime>* of the ModuleManager executable. The default *frameTime* is *0.01s*, making the manager run at 100Hz.

Example:

VTD 2.x:
1. open the file `simServer.xml` in *Data/Setups/Current/Config/SimServer*
2. add the parameter `-t` to the existing *moduleManager* entry's `cmdLine`

```
       <Process
         name="ModuleManager"
         auto="false"
         explicitLoad="false"
         path="$VI_CORE_DIR/ModuleManager"
         executable="moduleManager"
         cmdline="-f moduleManager.xml -t 0.2"
         affinitymask="0xFF"
         schedPolicy="SCHED_RR"
         schedPriority="20"
         useXterm="true"
         xtermOptions="-fg Black -bg Orange1 -geometry 80x10+1016+163"
         workDir="$VI_CURRENT_SETUP/Bin">
       </Process>
```

3. restart VTD (the simServer needs to be restarted so that it reads the new configuration file)

Important Note:

The default implementation of the *moduleManager* takes care that each message sent by the *TaskControl* is being handled. If your *moduleManager* is running at considerably lower frequency than the *TaskControl*, messages may "pile up" internally and you may even have to process older messages before getting to the latest one. You may force the *moduleManager* to forward only the latest received message to the plug-ins by providing the command line argument `-l` (work on **l** ast received frame only). Example:

```
       <Process
         name="ModuleManager"
         :
         cmdline="-f moduleManager.xml -t 0.2 -l"
         :
         workDir="$VI_CURRENT_SETUP/Bin">
       </Process>
```

VTD 1.x:

1. copy the `moduleManager` entry from *Data/Setups/Common/Config/Simulation/taskSettings.cfg* to your setup (i.e. to *Data/Setups/Current/Config/Simulation/taskSettings.cfg*).
2. add the `-t` command line argument, e.g. for 1kHz

```
# ......... MODULE MANAGER .........
#
if { test $TASK_TYPE = "moduleManager" } then

  set BIN_NAME      = moduleManager
  set WORK_DIR      = ( . )
  set OBJ_DIR       = ( $VI_CORE_DIR/ModuleManager )
  set CMD_ARGS      = ( -t 0.001 -f moduleManager.xml )
  set START_CMD     = ( source $ENV_SETTINGS ; cd $WORK_DIR ; $OBJ_DIR/$BIN_NAME $CMD_ARGS $RECORD_CMD ; sleep $XTERM_LIF
  set USE_XTERM     = true
  set XTERM_OPTIONS = ( $XTERM_OPTIONS_COMMON -bg Orange1 -geometry 80x10+1016+163 -title $TASK_TYPE )
endif
#
#
```

3. re-load all components

How can I change the network interface the ModuleManager is working on?

The ModuleManager may be configured to use an Ethernet interface different from eth0. In order to assign the device, use the command line argument *-i <deviceName>* of the ModuleManager executable. The default *deviceName* is *eth0*.

Example:

VTD 2.x:
1. open the file `simServer.xml` in *Data/Setups/Current/Config/SimServer*
2. add the parameter `-i` to the existing *moduleManager* entry's `cmdLine`

```
       <Process
         name="ModuleManager"
         auto="false"
         explicitLoad="false"
         path="$VI_CORE_DIR/ModuleManager"
         executable="moduleManager"
         cmdline="-f moduleManager.xml -i enps0"
         affinitymask="0xFF"
```

```
          schedPolicy="SCHED_RR"
          schedPriority="20"
          useXterm="true"
          xtermOptions="-fg Black -bg Orange1 -geometry 80x10+1016+163"
          workDir="$VI_CURRENT_SETUP/Bin">
        </Process>
```

3. restart VTD (the simServer needs to be restarted so that it reads the new configuration file)

How can I operate the Module Manager in fully frame-synchronous mode?

As described in the previous paragraph, you may influence the frequency of the Module Manager with the command line parameter `-t`. If you want the Module Manager to run in frame-synchonous mode with the TaskControl (i.e. only executing the internal update routines once after having received a complete RDB frame), then you should set the timing to `0.0`.

The configuration line given above will then read:

VTD 2.x:

```
        <Process
          name="ModuleManager"
          :
          cmdline="-f moduleManager.xml -t 0.0"
          :>
        </Process>
```

VTD 1.x:

```
    set CMD_ARGS      = ( -t 0.0 -f moduleManager.xml )
```

How can I run multiple instances of the Module Manager?

In order to run multiple instances of the moduleManager (e.g. for applying different frequencies to the plug-ins), you may run more than one instance in parallel, each with its own configuration file.

Example:

VTD 2.x:

1. open the file `simServer.xml` in *Data/Setups/Current/Config/SimServer*
2. copy the existing entry for the *ModuleManager*
3. rename the copy
4. provide a dedicated configuration file
5. optional: add the parameter `-t` to the command line of *ModuleManager2*

```
        <Process
          name="ModuleManager"
          auto="false"
          explicitLoad="false"
          path="$VI_CORE_DIR/ModuleManager"
          executable="moduleManager"
          cmdline="-f moduleManager.xml"
          affinitymask="0xFF"
          schedPolicy="SCHED_RR"
          schedPriority="20"
          useXterm="true"
          xtermOptions="-fg Black -bg Orange1 -geometry 80x10+1016+163"
          workDir="$VI_CURRENT_SETUP/Bin">
        </Process>

        <Process
          name="ModuleManager2"
          auto="false"
          explicitLoad="false"
          path="$VI_CORE_DIR/ModuleManager"
          executable="moduleManager"
          cmdline="-f moduleManager2.xml -t 1.0 -l"
          affinitymask="0xFF"
          schedPolicy="SCHED_RR"
          schedPriority="20"
          useXterm="true"
          xtermOptions="-fg Black -bg Orange1 -geometry 80x10+1016+326"
          workDir="$VI_CURRENT_SETUP/Bin">
        </Process>
```

6. restart VTD (the simServer needs to be restarted so that it reads the new configuration file)

How can I define a dynamics plug-in which controls a vehicle (i.e. EGO vehicle) via the VIRES driver?

If your own vehicle shall be controlled by the VIRES driver (i.e. the one which is also used for the traffic vehicles) and shall use the VIRES vehicle dynamics (again, the one which is also used by the traffic vehicles) you have to perform the following steps:

1. Define the desired vehicle as *extern* in the ScenarioEditor

2. Configure the ModuleManager using its configuraton file *moduleManager.xml* (typically at *Data/Projects/Current/Config/ModuleManager*). Example:

```
<DynamicsPlugin name="viTrafficDyn">
    <Load     lib="libModuleTrafficDyn.so" path=""/>
    <Player   default="true" />
    <Debug    enable="false"
              dynInput="true"
              dynOutput="true"
              CSV="false"
              packages="true"/>
</DynamicsPlugin>
```

The identification of the vehicle which shall be controlled by the ModuleManager can be done in two ways:

1. `<Player default="true"/>` will assign the first external player in the scenario to the plug-in
2. `<Player name="Iknowthename"/>` will assign the external player whose name matches the given one to the plug-in

How is the position of a sensor computed?

A sensor is attached to a player whose name may be explicitly defined or who is identified as the first external player in a scenario.

Example:

```
<Sensor name="perfect" type="video">
    ...
    <Player   default="true" />
    <Position dx="3.5" dy="0.0" dz="0.5" dhDeg="0.0" dpDeg="0.0" drDeg="0.0" />
    ...
</Sensor>
```

The identification of the vehicle which shall carry the sensor can be done in two ways:

1. `<Player default="true"/>` will assign the sensor to the first external player in the scenario
2. `<Player name="Iknowthename"/>` will assign the sensor to the player whose name matches the given one
   Attention: up to VTD.1.1.2 you also need to set the attribute `default="false"`

The position of the sensor (tag `<Position>` in the above example) will be computed relative to the carrying vehicle's reference point (typically center of rear axle on ground level) by first applying the translations *dx*, *dy* and *dz* in player co-ordinates and then rotating the sensor by *dhDeg* (z-axis), *dpDeg* (y*-axis) and *drDeg* (x**-axis). Each rotation is performed in the system resulting from the previous rotation.

How can I disable individual plug-ins?

Plug-ins of the ModuleManager may be enabled/disabled individually during runtime. Use one of the following commands:

```
<DynamicsPlugin name="viTrafficDyn" enable="false" />
```

```
<Sensor name="perfect" enable="false" />
```

Or use the generic command

```
<Plugin name="name of the plugin" enable="[true | false]">
```

How do I query the list of loaded plug-ins?

availability: VTD 2.0.2+

You may query the list of loaded plug-ins from the ModuleManager with the following command:

```
<Query entity="moduleManager"><Plugin/></Query>
```

The result will be similar to the following:

```
<Reply entity="moduleManager">
  <Sensor name="DefaultPerfectSensor" />
  <DynamicsPlugin name="viTrafficDyn" />
  <DynamicsPlugin name="viTrafficDynComplex" />
</Reply>
```

Performance Monitoring

Legacy Mechanism

Starting with VTD 2.0, the debug options which are set in the moduleManager's configuration file (typically *Data/Setups/Current/Config/ModuleManager /moduleManager.xml*) may hold an additional attribute `performance`.

Example:

```
<RDB>
    <Port name="RDBraw" number="48190" type="TCP" />
```

```
</RDB>
<Debug    enable="true"
          :
          performance="true" />
:
```

If set to *true* (and also setting `enable` to `true`), the moduleManager will provide shell output with the computation time per plug-in and for the entire frame (the latter only if the execution time exceeds 100us).

Example:

```
NOTICE (moduleMgr): "Controller::updateWrapper: time for executing plugin <perfect>: 0.001ms"
NOTICE (moduleMgr): "Controller::updateWrapper: time for executing plugin <viTrafficDyn>: 0.001ms"
NOTICE (moduleMgr): "Controller::updateWrapper: time for executing plugin <perfect>: 0.000ms"
NOTICE (moduleMgr): "Controller::updateWrapper: time for executing plugin <viTrafficDyn>: 0.001ms"
NOTICE (moduleMgr): "Controller::updateWrapper: time for executing plugin <perfect>: 0.001ms"
NOTICE (moduleMgr): "Controller::updateWrapper: time for executing plugin <viTrafficDyn>: 0.001ms"
NOTICE (moduleMgr): "Controller::updateWrapper: time for executing plugin <perfect>: 0.150ms"
NOTICE (moduleMgr): "Controller::updateWrapper: time for executing plugin <viTrafficDyn>: 0.035ms"
main: update duration = 0.39 [ms], average duration = 0.66 [ms]
```

Advanced Mechanism

availability: VTD 2.0.3+

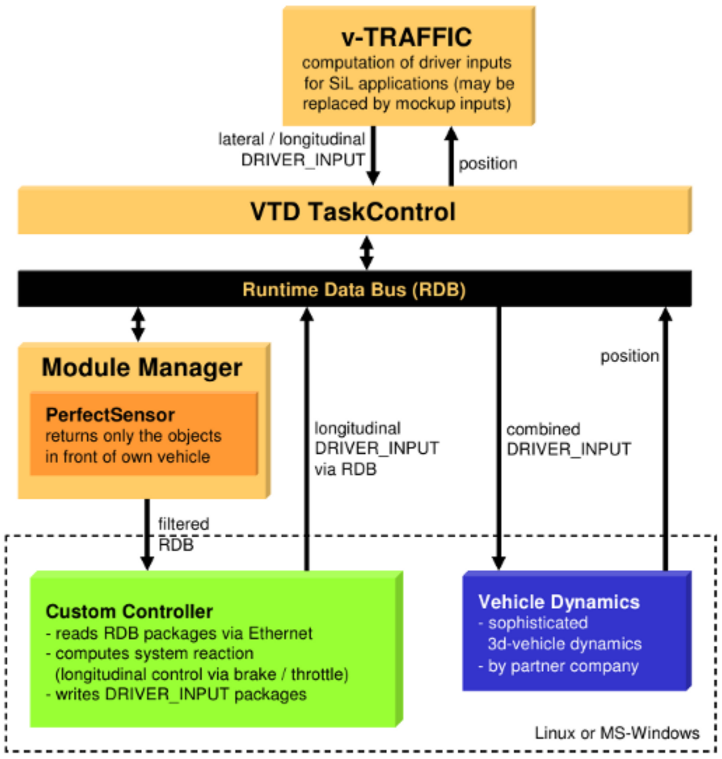
see Performance Monitoring

Note: the tag `<RealtimeMonitor>` may be given in the configuration file `moduleManager.mxl`

Plug-Ins

Sensor plug-ins are used for the extraction of data from the virtual world within a given sub-space which is usually connected to the own vehicle. The standard sensors work like filter which are adding some information about occlusion etc.

The following pictures shows a typical sensor configuration for e.g. an ACC test case.


v-TRAFFIC
computation of driver inputs for SiL applications (may be replaced by mockup inputs)
lateral / longitudinal DRIVER_INPUT
position
VTD TaskControl
Runtime Data Bus (RDB)
Module Manager
PerfectSensor
returns only the objects in front of own vehicle
longitudinal DRIVER_INPUT via RDB
combined DRIVER_INPUT
position
filtered RDB
Custom Controller
- reads RDB packages via Ethernet
- computes system reaction (longitudinal control via brake / throttle)
- writes DRIVER_INPUT packages
Vehicle Dynamics
- sophisticated 3d-vehicle dynamics
- by partner company
Linux or MS-Windows

Sensor plug-ins have to run within the ModuleManager. One ModuleManager may handle multiple sensors. Also, different instances of the ModuleManager may run in parallel. The sensor plug-ins may be differently configured instances of the "PerfectSensor" or other sensors provided with VTD or they may be completely individual plug-ins written by the users.

Sensor Plug-ins provided with VTD

PerfectSensor

Overview

plugin: `libModulePerfectSensor.so`.

This sensor may be positioned anywhere on the carrier (e.g. own vehicle). It detects objects within its pyramid-shaped frustum whose detection range is limited by near- and far-clipping plane.

The dimensions of the sensor frustum may be defined via the GUI or in the configuration file of the ModuleManager (`Data/Projects/Current/Config/ModuleManager/moduleManager.xml`). If you don't have a local copy of the ModuleManager configuration file in your project, then copy it from the distribution (`Data/Distros/Current/Config/ModuleManager/moduleManager.xml`). Do **NOT** modify the configuration file in the distribution.

In the ModuleManager configuration file, you may configure multiple sensors. Make sure that each sensor in the file has a unique name and a unique number of the output port. In the following example, three perfect sensors are provided:

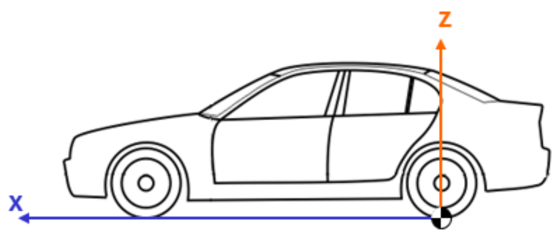
```
<Sensor name="perfectFront" type="video">
    <Load     lib="libModulePerfectSensor.so" path="" persistent="true" />
    <Frustum  near="0.0" far="50.0" left="10.0" right="10.0" bottom="3.0" top="3.0" />
    <Cull     maxObjects="5" enable="true" />
    <Port     name="RDBout" number="48195" type="UDP" sendEgo="true" />
    <Player   default="true" />
    <Position dx="3.5" dy="0.0" dz="0.5" dhDeg="0.0" dpDeg="0.0" drDeg="0.0" />
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Filter   objectType="trafficSign"/>
    <Filter   objectType="obstacle"/>
    <Debug    enable="false" />
</Sensor>

<Sensor name="perfectLeft" type="video">
    <Load     lib="libModulePerfectSensor.so" path="" persistent="true" />
    <Frustum  near="0.0" far="50.0" left="10.0" right="10.0" bottom="3.0" top="3.0" />
    <Cull     maxObjects="5" enable="true" />
    <Port     name="RDBout" number="48196" type="UDP" sendEgo="true" />
    <Player   default="true" />
    <Position dx="3.5" dy="0.0" dz="0.5" dhDeg="90.0" dpDeg="0.0" drDeg="0.0" />
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Filter   objectType="trafficSign"/>
    <Filter   objectType="obstacle"/>
    <Debug    enable="false"  />
</Sensor>

<Sensor name="perfectRoad" type="video">
    <Load     lib="libModulePerfectSensor.so" path="" persistent="true" />
    <Frustum  near="0.0" far="50.0" left="10.0" right="10.0" bottom="3.0" top="3.0" />
    <Cull     maxObjects="5" enable="true" />
    <Port     name="RDBout" number="48197" type="TCP" sendEgo="true" />
    <Player   default="true" />
    <Position dx="3.5" dy="0.0" dz="0.5" dhDeg="0.0" dpDeg="0.0" drDeg="0.0" />
    <Filter   objectType="roadInfo"/>
    <Filter   objectType="laneInfo"/>
    <Filter   objectType="roadMarks" tesselate="true"/>
    <Debug    enable="false" />
</Sensor>
```

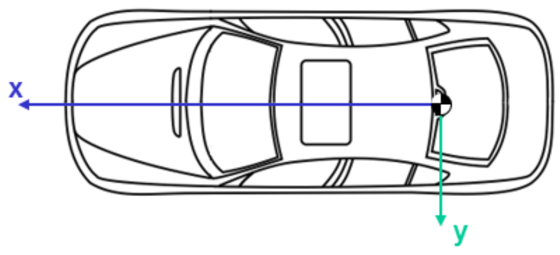
The sensor "perfectFront" reads the RDB data stream from the TaskControl and keeps players which are (fully or partially) contained in its cone (which is oriented in forward direction). All other players are discarded. It also detects static obstacles and traffic signs from the information provided in the OpenDRIVE road database. The sensor "perfectLeft" will do the same but its orientation is to the left side of the own vehicle. The sensor "perfectRoad" provides road information according to a pre-defined scheme. Its cone is not actually used, but it could of course be used if filtering for objects of the above types was enabled.

After detecting the relevant objects (or calculating the respective information), each sensor will compose an RDB output data package containing the relevant information (object lists etc.). The output will be sent via the ports defined in the `<Port>` section of each sensor. Each sensor will open its own output port (the first two using UDP and the last one TCP/IP). User's systems shall attach to these ports. If a TCP/IP port is specified, then the sensor will be the server and user's components shall connect as clients.

Co-ordinate Systems

The positions of detected objects are given in a "USK" co-ordinate system per default. The origin and orientation of this system are identical to the vehicle's co-ordinate system.





The user may specify other co-ordinate systems for the output of objects. This can be done using the `<Origin>` tag in the sensor configuration:

```
<Origin type="{USK|inertial|sensor|relative|gps|road}" dx=... dy=... dz=... dhDeg=... dpDeg=... drDeg=.../>
```

Notes:

- the attributes `dx, dy, dz, dhDeg, dpDeg, drDeg` may be used in connection with the type *relative* only. Using these offsets, you may set the reference system's origin anywhere relative to the sensor's carrier.
- the type `road` must only be used for sensors derived from the perfect sensor which are internally already computing road / track co-ordinates for each object; the conversion is not being taken care of by the base class.

For more information about the co-ordinate systems, please refer to General_Definitions

Geo Co-ordinates:

Per default, i.e. on the "main" RDB connection of the TaskControl, no data will be sent as Geo Co-ordinates (see also General Definitions. In order to retrieve data in GPS co-ordinates, you have to set up a given sensor accordingly by specifying the `<Origin type="gps"/>` tag in the `moduleManager.xml` configuration file. A **special case** is the retrieval of all dynamic elements' co-ordinates as GPS data. In this case, just setup a sensor as given in the following example (add these lines to your `moduleManager.xml` configuration file and adapt the output port accordingly):

```
<Sensor name="GPSSensor" type="video">
    <Load     lib="libModulePerfectSensor.so" path="" persistent="true" />
    <Cull     enable="false" />
    <Port     name="RDBout" number="48195" type="TCP" sendEgo="true" />
    <Player   default="true" />
    <Origin   type="gps"/>
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Debug    enable="false" />
</Sensor>
```

Occlusion Information

The perfect sensor will compute the occlusion of objects by each other and it will provide this information in the sensor object's data. The occlusion is given in a range from 0 (un-occluded) to 127 (completely occluded) or -1 (no data available), respectively. The user may also specify that objects with an occlusion higher than a user-defined threshold (normalized from 0.0 to 1.0) shall not be sent via the sensor's output. The corresponding attribute in the sensor configuration file (typically `Data/Projects/Current/Config/ModuleManager/moduleManager.xml`) is `maxOcclusion` in the tag `<Cull>`.

Example:

`<Cull maxObjects="5" enable="true" maxOcclusion="0.3"/>`

In addition to the mere filtering, the user may also request the sensor's internal occlusion matrix of a given object for further analysis.

Configuration example:

```
<Sensor name="perfect" type="video">
    <Load     lib="libModulePerfectSensor.so" path="" persistent="true" />
    :
    <Output   sendOcclusionMatrix="true"/>
    :
</Sensor>
```

The occlusion matrix will be sent as an RDB package of type `RDB_PKG_ID_OCCLUSION_MATRIX` which is actually an `RDB_IMAGE_t` with pixelSize of 32bit and RED32 pixel format (i.e. it is an array of 10 x 10 integers). The matrix spreads evenly over the horizontal and vertical field-of-view of a sensor. A cell's value of 0 means that it contains no object within the sensor range, a value of 1 means that the target object occupies this cell and a value greater than 1 means that the cell is also occluded by additional objects (with (n - 1) being the number of objects also contained in the cell). The attribute `id` provided with the occlusion matrix is the unique ID of the object to which the matrix refers.

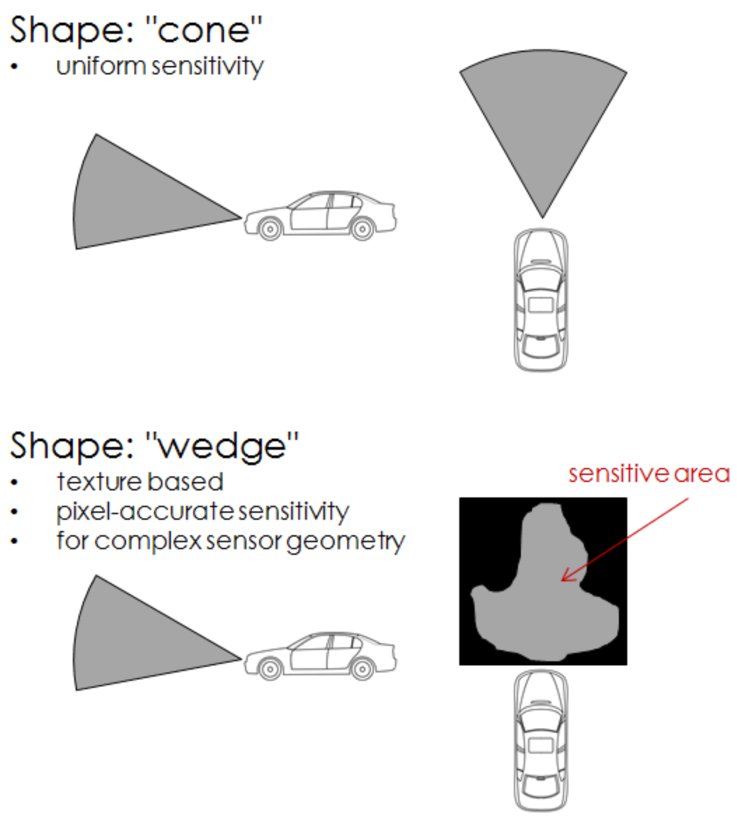
Example (output of RDB sniffer):

```
RDBSniffer::printMessage: ---- full info ----- BEGIN
  message: version = 0x0117, simTime = 13.233, simFrame = 795, headerSize = 24, dataSize = 8104
    entry: pkgId = 39 (RDB_PKG_ID_OCCLUSION_MATRIX), headersize = 16, dataSize = 2160, elementSize = 432, noElements = 5, flag
      id               = 2
      width            = 10
      height           = 10
      pixelSize        = 32
      pixelFormat      = 32
      cameraId         = 0
      imgSize          = 400
      color            = 0/0/0/0
    0    0    1    1    1    1    1    1    0    0
    0    0    1    1    1    1    1    1    0    0
    0    0    1    1    1    1    1    1    0    0
    0    0    1    1    1    1    1    1    0    0
    0    0    1    1    1    1    1    1    0    0
    0    0    1    1    1    1    1    1    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
```

Custom Sensor Shapes

availability: VTD 2.0.2+

Unless otherwise specified, the sensor's frustum will be **cone**-shaped. For the description of more complex frusta, the user may specify the shape of the sensor's *horizontal cross section* using a texture (.png file) which is applied to a **wedge**-shaped base frustum. The following image illustrates these alternatives:



The frustm definition in the ModuleManager configuration file (typically `Data/Projects/Current/Config/ModuleManager/moduleManager.xml`) is as follows:

cone sensor

```
<Sensor name="perfect" type="video">
```

```
    <Load     lib="libModulePerfectSensor.so" ...... />
    :
    <Frustum  near="0.0" far="100" left="30" right="30" bottom="2.0" top="2.0" />
    :
</Sensor>
```

here, the frustum parameters have the following meaning:

near: near clipping plane in [m]
far: far clipping plane in [m]
left: cone angle to the left in [deg]
right: cone angle to the right in [deg]
bottom: cone angle to the bottom in [deg]
top: cone angle to the top in [deg]

wedge sensor

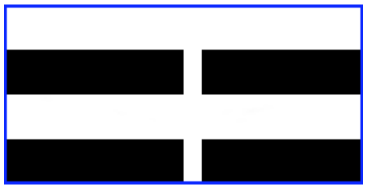
```
<Sensor name="perfect" type="video">
    <Load     lib="libModulePerfectSensor.so" ...... />
    :
    <Frustum  near="0.0" far="100" left="30" right="30" bottom="2.0" top="2.0" shape="wedge" file="/tmp/myFrust.png" />
    :
</Sensor>
```

here, the frustum parameters have the following meaning:

near: near clipping plane in [m]
far: far clipping plane in [m]
left: wedge's left border **in [m]**
right: wedge's right border **in [m]**
bottom: wedge angle to the bottom in [deg]
top: wedge angle to the top in [deg]
shape: must be "wedge"
file: path to the texture file describing the custom shape

texture file

Example:



The texture file describes the sensitivity of individual parts of the wedge in the wedge's horizontal cross section. The texture is *aligned* as follows:

- left edge texture = left edge wedge
- right edge texture = right edge wedge
- bottom edge texture = near edge wedge
- top edge texture = far edge wedge

The texture will be distributed uniformly over the wedge's horizontal cross section.

The *coding* of the texture content is:

- white: sensitive area
- black: insensitive area
- blue: border, for illustration only, not to be used

Grey levels indicating varying sensitivity levels will be introduced in an upcoming version of the sensor.

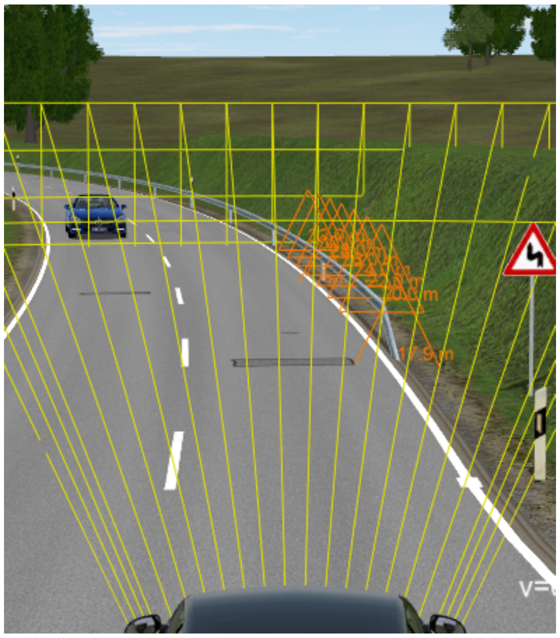
In the texture example above, the following interpretation applies:

- the farthest 25% of sensor range are fully sensitive
- between 50% and 75% of range, only a small longitudinal area is sensitive
- between 25% and 50% of range, the whole area is sensitive
- between 0% and 25% of range, only a small longitudinal area is sensitive

Note: the visualization of a wedge-shaped sensor in the image generator is not yet fully implemented. Instead, the cone-shaped sensor will be shown. This is a known deficiency of the current implementation.

Detection of Continuous Objects

For the detection of continuous objects (e.g. railings), the perfect sensor performs an internal tesselation and reports all tesselated element that fit within its sensor cone:



The tesselation parameters may be set in the sensor configuration section of the moduleManager's configuration file. Example:

```
<Sensor name="DefaultPerfectSensor" type="video">
    :
    <Database resolveRepeatedObjects="true" continuousObjectTesselation="2.0" />
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Filter   objectType="obstacle"/>
    :
</Sensor>
```

Under the tag `<Database>`, the user may specify whether repeated object shall be resolved (i.e. instantiated) and what tesselation step size in `[m]` shall be used for continuous objects.

On the sensor's RDB output port, there will be - among others - one package of type **SENSOR_OBJECT_t** and one package of type **OBJECT_STATE_t** for each detected element. The output data corresponding to the above image is:

SENSOR_OBJECT:

```
RDBSniffer::printMessage: ---- full info ----- BEGIN
  message: version = 0x011a, simTime = 82.747, simFrame = 4966, headerSize = 24, dataSize = 7588
    entry: pkgId = 17 (RDB_PKG_ID_SENSOR_OBJECT), headersize = 16, dataSize = 760, elementSize = 76, noElements = 10, flags =
      category = 1
      type     = 1
      flags    = 0x3
      id       = 38
      sensorId = 3
      dist     = 11.307
      sensorPos
          x/y/z = 10.936 / 2.872 / -0.007
          h/p/r = -3.139 / -0.001 / 0.000
          flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
      occlusion = 0

      category = 1
      type     = 1
      flags    = 0x3
      id       = 26
      sensorId = 3
      dist     = 17.209
      sensorPos
          x/y/z = 16.900 / 3.243 / -0.017
          h/p/r = -3.106 / -0.002 / 0.000
          flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
      occlusion = 84

      category = 5
      type     = 0
      flags    = 0x3
      id       = 1321
      sensorId = 3
      dist     = 17.948
```

```
sensorPos
    x/y/z = 17.868 / -1.673 / -0.229
    h/p/r = -3.074 / 0.000 / 0.000
    flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
occlusion = 0

category = 5
type     = 0
flags    = 0x3
id       = 1321
sensorId = 3
dist     = 19.953
sensorPos
    x/y/z = 19.889 / -1.528 / 0.467
    h/p/r = -3.066 / 0.000 / 0.000
    flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
occlusion = 0

category = 5
type     = 0
flags    = 0x3
id       = 1279
sensorId = 3
dist     = 19.953
sensorPos
    x/y/z = 19.889 / -1.528 / 0.467
    h/p/r = -3.066 / 0.000 / 0.000
    flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
occlusion = 127

category = 5
type     = 0
flags    = 0x3
id       = 1279
sensorId = 3
dist     = 21.957
sensorPos
    x/y/z = 21.909 / -1.367 / 0.462
    h/p/r = -3.058 / 0.000 / 0.000
    flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
occlusion = 127

category = 5
type     = 0
flags    = 0x3
id       = 1279
sensorId = 3
dist     = 23.962
sensorPos
    x/y/z = 23.928 / -1.191 / 0.456
    h/p/r = -3.051 / 0.000 / 0.000
    flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
occlusion = 127

category = 5
type     = 0
flags    = 0x3
id       = 1279
sensorId = 3
dist     = 25.969
sensorPos
    x/y/z = 25.946 / -0.999 / 0.450
    h/p/r = -3.043 / 0.000 / 0.000
    flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
occlusion = 127

category = 5
type     = 0
flags    = 0x3
id       = 1279
sensorId = 3
dist     = 27.977
sensorPos
    x/y/z = 27.962 / -0.792 / 0.444
    h/p/r = -3.035 / 0.000 / 0.000
    flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
occlusion = 63

category = 5
type     = 0
flags    = 0x3
id       = 1279
sensorId = 3
dist     = 29.985
sensorPos
```

```
          x/y/z = 29.976 / -0.569 / 0.437
          h/p/r = -3.028 / 0.000 / 0.000
          flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
      occlusion = 127
RDBSniffer::printMessage: ---- full info ----- END
```

OBJECT_STATE (please note the name of each object which states that it's actually a **RAILING**)

```
RDBSniffer::printMessage: ---- full info ----- BEGIN
  message: version = 0x011a, simTime = 166.177, simFrame = 9972, headerSize = 24, dataSize = 7588
    entry: pkgId =  9 (RDB_PKG_ID_OBJECT_STATE), headersize = 16, dataSize = 896, elementSize = 112, noElements = 8, flags = 0
      id       = 1321
      category = RDB_OBJECT_CATEGORY_COMMON
      type     = RDB_OBJECT_TYPE_PLAYER_NONE
      visMask  = 0x1
      name     = RAILING_STANDARD
      geometry
          dim x / y / z = 2.000 / 0.080 / 0.210
          off x / y / z = 0.000 / 0.000 / 0.000
      position
          x/y/z = 17.868 / -1.673 / -0.229
          h/p/r = -3.074 / 0.000 / 0.000
          flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
      parent     = 0
      cfgFlags   = 0x0
      cfgModelId = 0

      id       = 1321
      category = RDB_OBJECT_CATEGORY_COMMON
      type     = RDB_OBJECT_TYPE_PLAYER_NONE
      visMask  = 0x1
      name     = RAILING_STANDARD
      geometry
          dim x / y / z = 2.000 / 0.080 / 0.210
          off x / y / z = 0.000 / 0.000 / 0.000
      position
          x/y/z = 19.889 / -1.528 / 0.467
          h/p/r = -3.066 / 0.000 / 0.000
          flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
      parent     = 0
      cfgFlags   = 0x0
      cfgModelId = 0

      id       = 1279
      category = RDB_OBJECT_CATEGORY_COMMON
      type     = RDB_OBJECT_TYPE_PLAYER_NONE
      visMask  = 0x1
      name     = RAILING_STANDARD
      geometry
          dim x / y / z = 2.000 / 0.080 / 0.210
          off x / y / z = 0.000 / 0.000 / 0.000
      position
          x/y/z = 19.889 / -1.528 / 0.467
          h/p/r = -3.066 / 0.000 / 0.000
          flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
      parent     = 0
      cfgFlags   = 0x0
      cfgModelId = 0

      id       = 1279
      category = RDB_OBJECT_CATEGORY_COMMON
      type     = RDB_OBJECT_TYPE_PLAYER_NONE
      visMask  = 0x1
      name     = RAILING_STANDARD
      geometry
          dim x / y / z = 2.000 / 0.080 / 0.210
          off x / y / z = 0.000 / 0.000 / 0.000
      position
          x/y/z = 21.909 / -1.367 / 0.462
          h/p/r = -3.058 / 0.000 / 0.000
          flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
      parent     = 0
      cfgFlags   = 0x0
      cfgModelId = 0

      id       = 1279
      category = RDB_OBJECT_CATEGORY_COMMON
      type     = RDB_OBJECT_TYPE_PLAYER_NONE
      visMask  = 0x1
      name     = RAILING_STANDARD
      geometry
          dim x / y / z = 2.000 / 0.080 / 0.210
          off x / y / z = 0.000 / 0.000 / 0.000
      position
```

```
            x/y/z = 23.928 / -1.191 / 0.456
            h/p/r = -3.051 / 0.000 / 0.000
            flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
        parent     = 0
        cfgFlags   = 0x0
        cfgModelId = 0

        id       = 1279
        category = RDB_OBJECT_CATEGORY_COMMON
        type     = RDB_OBJECT_TYPE_PLAYER_NONE
        visMask  = 0x1
        name     = RAILING_STANDARD
        geometry
            dim x / y / z = 2.000 / 0.080 / 0.210
            off x / y / z = 0.000 / 0.000 / 0.000
        position
            x/y/z = 25.946 / -0.999 / 0.450
            h/p/r = -3.043 / 0.000 / 0.000
            flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
        parent     = 0
        cfgFlags   = 0x0
        cfgModelId = 0

        id       = 1279
        category = RDB_OBJECT_CATEGORY_COMMON
        type     = RDB_OBJECT_TYPE_PLAYER_NONE
        visMask  = 0x1
        name     = RAILING_STANDARD
        geometry
            dim x / y / z = 2.000 / 0.080 / 0.210
            off x / y / z = 0.000 / 0.000 / 0.000
        position
            x/y/z = 27.962 / -0.792 / 0.444
            h/p/r = -3.035 / 0.000 / 0.000
            flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
        parent     = 0
        cfgFlags   = 0x0
        cfgModelId = 0

        id       = 1279
        category = RDB_OBJECT_CATEGORY_COMMON
        type     = RDB_OBJECT_TYPE_PLAYER_NONE
        visMask  = 0x1
        name     = RAILING_STANDARD
        geometry
            dim x / y / z = 2.000 / 0.080 / 0.210
            off x / y / z = 0.000 / 0.000 / 0.000
        position
            x/y/z = 29.976 / -0.569 / 0.437
            h/p/r = -3.028 / 0.000 / 0.000
            flags = 0x3, type = RDB_COORD_TYPE_USK, system = 0
        parent     = 0
        cfgFlags   = 0x0
        cfgModelId = 0
```

Extended PerfectSensor

Overview

plugin: `libModulePerfectSensorExt.so`.

This sensor is an extended version of the *PerfectSensor* plugin (see previous chapter). The extensions are:

- the detection point of an object based on its bounding box may be set to
  - nearest point on any bottom edge pointing to the sensor
  - center point of the bounding box's bottom plane
  - front/rear bottom edge
  - edge centers (only VTD 2.1+)

In order to configure this behavior, the sensor configuration block (e.g. in the ModuleManager configuration file) may be extended as follows:

```
<Sensor name="perfectFront" type="video">
    <Load     lib="libModulePerfectSensorExt.so" path="" persistent="true" />
    :
    <Config useNearestPoint="true" useCenter="false" useFrontRear="false" useEdgeCenters="false"/>
    :
</Sensor>
```

CameraSensor

plugin: `libModuleCameraSensor.so`.

The CameraSensor resembles a sensor which is located at the camera position and uses the same intrinsic parameters. The primary output of the camera sensor is in screen co-ordinates, although some data may also be provided in inertial or USK co-ordinate systems. The flags of the respective position elements (`RDB_COORD_t`) are set accordingly.

The CameraSensor has primarily been designed for the detection of light sources and traffic signs. It works in two stages

1. Detection like a perfect sensor - this provides the basic object data of vehicles, pedestrians etc.
2. Additional detection AND occlusion calculation for the following types of objects:
    - traffic signs
    - vehicle lights (headlights, rear lights)
    - street lamps
    - common obstacles (e.g. houses) which may occlude the former objects

In order for the first stage to work correctly, make sure that the basic cone parameters of the sensor are set correctly and correspond closely to the camera parameters. The positions of all objects are returned as bounding boxes in screen space with the position co-ordinate itself being the origin of a rectangle whose dimensions are given in the geo member of an object of type `RDB_OBJECT_STATE_t`.

The object lists are sent by the CameraSensor via its TCP or UDP output port. Each detected object is contained in a structure of type `RDB_OBJECT_STATE_BASE_t`. The following fields are used for the distinction between the different object types:

```
Players
- objState.base.category = RDB_OBJECT_CATEGORY_PLAYER
- objState.base.type     = [RDB_OBJECT_TYPE_PLAYER_BIKE, RDB_OBJECT_TYPE_PLAYER_PEDESTRIAN,
                            RDB_OBJECT_TYPE_PLAYER_MOTORBIKE, RDB_OBJECT_TYPE_PLAYER_CAR]

traffic signs
- objState.base.category = RDB_OBJECT_CATEGORY_NONE
- objState.base.type     = RDB_OBJECT_TYPE_TRAFFIC_SIGN

vehicle lights:
- objState.base.category = RDB_OBJECT_CATEGORY_LIGHT_POINT
- objState.base.type     = RDB_OBJECT_TYPE_HEADLIGHT

street lamps:
- objState.base.category = RDB_OBJECT_CATEGORY_LIGHT_POINT
- objState.base.type     = RDB_OBJECT_TYPE_STREET_LAMP

common object (obstacle)
- objState.base.category = RDB_OBJECT_CATEGORY_COMMON
- objState.base.type     = RDB_OBJECT_TYPE_NONE

for all of the above mentioned objects, the following flags are set:
- objState.base.pos.type  = RDB_COORD_TYPE_WINDOW
- objState.base.pos.flags = RDB_COORD_FLAG_POINT_VALID
```

Additional information of the following types which applies to some of the detected objects is forwarded unaltered by the sensor after it has been received via the RDB data stream coming from the TC:

- `RDB_PKG_ID_OBJECT_CFG`
- `RDB_PKG_ID_VEHICLE_SYSTEMS`
- `RDB_PKG_ID_ROAD_POS`

If the forwarding of Ego information is enabled in the sensor setup, then the following data of the ownship will be added unaltered to the data stream:

- `RDB_PKG_ID_ROAD_STATE`
- `RDB_PKG_ID_OBJECT_STATE`
- `RDB_PKG_ID_OBJECT_CFG`
- `RDB_PKG_ID_ENGINE`
- `RDB_PKG_ID_DRIVETRAIN`
- `RDB_PKG_ID_WHEEL`
- `RDB_PKG_ID_VEHICLE_SYSTEMS`
- `RDB_PKG_ID_VEHICLE_SETUP`
- `RDB_PKG_ID_ROAD_POS`
- `RDB_PKG_ID_ROADMARK`
- `RDB_PKG_ID_LANE_INFO`

The information about the sensor itself (contained in a package of type `RDB_SENSOR_STATE_t`) complements the data stream.

CrashSensor

plugin: `libModuleCrashSensor.so`.

The CrashSensor detects crashes between the bounding box of the sensor's owner and the bounding boxes of surrounding players, objects and static obstacles (from OpenDRIVE database).

If a crash is detected, it will issue an SCP message.

An example for the sensor configuration is:

```
<Sensor name="crash" type="video">
    <Load     lib="libModuleCrashSensor.so" path="" persistent="true" />
    <Player   default="true" />
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Filter   objectType="trafficSign"/>
    <Filter   objectType="obstacle"/>
    <Debug    enable="false"/>
```

```
</Sensor>
```

The crash messages are formatted like in the following examples:

Collision with moved object

```
<Info>
  <Collision>
    <Object name="Own" id="1" category="RDB_OBJECT_CATEGORY_PLAYER" type="RDB_OBJECT_TYPE_PLAYER_CAR">
      <PosInertial x="1086.688" y="1472.439" z="10.234" h="1.571" p="0.000" r="0.000"/>
      <SpeedInertial x="0.000" y="2.778" z="-0.000" h="-0.000" p="-0.000" r="0.000"/>
      <BoundingBox dimX="2.375" dimY="0.899" dimZ="0.734"/>
    </Object>
    <Object name="FastCar" id="2" category="RDB_OBJECT_CATEGORY_PLAYER" type="RDB_OBJECT_TYPE_PLAYER_CAR">
      <PosInertial x="1086.769" y="1476.013" z="10.260" h="1.571" p="0.000" r="0.000"/>
      <SpeedInertial x="0.000" y="0.000" z="0.000" h="0.000" p="0.000" r="0.000"/>
      <BoundingBox dimX="1.257" dimY="0.740" dimZ="0.760"/>
    </Object>
  </Collision>
</Info>
```

Collision with OpenDRIVE object

```
<Info>
  <Collision>
    <Object name="Own" id="1" category="RDB_OBJECT_CATEGORY_PLAYER" type="RDB_OBJECT_TYPE_PLAYER_CAR">
      <PosInertial x="1086.688" y="1482.995" z="10.234" h="1.571" p="0.000" r="0.000"/>
      <SpeedInertial x="0.000" y="2.778" z="-0.000" h="-0.000" p="-0.000" r="0.000"/>
      <BoundingBox dimX="2.375" dimY="0.899" dimZ="0.734"/>
    </Object>
    <Object name="RdMiscRoadDamage05.flt" id="1594" category="RDB_OBJECT_CATEGORY_OPENDRIVE" type="0">
      <PosInertial x="1086.387" y="1485.750" z="9.520" h="1.571" p="0.000" r="0.000"/>
      <BoundingBox dimX="0.480" dimY="0.610" dimZ="0.000"/>
    </Object>
  </Collision>
</Info>
```

Collision with static object

```
<Info>
  <Collision>
    <Object name="Own" id="1" category="RDB_OBJECT_CATEGORY_PLAYER" type="RDB_OBJECT_TYPE_PLAYER_CAR">
      <PosInertial x="1086.688" y="1484.884" z="10.234" h="1.571" p="0.000" r="0.000"/>
      <SpeedInertial x="0.000" y="2.778" z="-0.000" h="-0.000" p="-0.000" r="0.000"/>
      <BoundingBox dimX="2.375" dimY="0.899" dimZ="0.734"/>
    </Object>
    <Object name="myCube" id="6" category="RDB_OBJECT_CATEGORY_COMMON" type="RDB_OBJECT_TYPE_PLAYER_NONE">
      <PosInertial x="1085.509" y="1487.741" z="10.000" h="0.000" p="0.000" r="0.000"/>
      <SpeedInertial x="0.000" y="0.000" z="0.000" h="0.000" p="0.000" r="0.000"/>
      <BoundingBox dimX="0.500" dimY="0.500" dimZ="0.500"/>
    </Object>
  </Collision>
</Info>
```

Collision with pedestrian

```
<Info>
  <Collision>
    <Object name="Own" id="1" category="RDB_OBJECT_CATEGORY_PLAYER" type="RDB_OBJECT_TYPE_PLAYER_CAR">
      <PosInertial x="1086.688" y="1493.884" z="10.234" h="1.571" p="0.000" r="0.000"/>
      <SpeedInertial x="0.000" y="2.778" z="-0.000" h="-0.000" p="-0.000" r="0.000"/>
      <BoundingBox dimX="2.375" dimY="0.899" dimZ="0.734"/>
    </Object>
    <Object name="fred" id="5" category="RDB_OBJECT_CATEGORY_PLAYER" type="RDB_OBJECT_TYPE_PLAYER_PEDESTRIAN">
      <PosInertial x="1086.283" y="1496.414" z="10.156" h="0.000" p="0.000" r="0.000"/>
      <SpeedInertial x="0.000" y="0.000" z="0.000" h="0.000" p="0.000" r="0.000"/>
      <BoundingBox dimX="0.119" dimY="0.257" dimZ="0.656"/>
    </Object>
  </Collision>
</Info>
```

In addition to the standard crash message, the user may also define a user-specific SCP message that is issued upon detection of a crash. For this, the tag `<Output scpUserMessage="true">` must be provided in the sensor definition and the associated SCP message must be provided as CDATA. **Note:** this feature is only available in VTD 1.4 and higher.

Example (plays a sound and shows a popup message in the GUI upon crash detection):

```
<Sensor name="crash" type="video">
    <Load     lib="libModuleCrashSensor.so" path="" persistent="true" />
    <Output   scpUserMessage="true">
        <![CDATA[<Sound name="mySound"><Start id="0"/></Sound><Info><Message text="You crashed!" popup="true"/></Info>]]>
    </Output>
    <Player   default="true" />
    <Debug    enable="false"/>
</Sensor>
```

Object Filtering

The crash sensor may be configured to react on certain **object types** only (available in VTD 1.4+). Per default, all possible collisions will be detected. Upon definition of the first filter, the user has to provide filter entries for all object types that are of interest. The filter definitions have to be provided within the `<Sensor>` definition. Two different filter types are available:

Standard VTD objects (e.g. players)

> *Each filter definition consists of two attributes: rdbCategory and rdbType. The valid values for these attributes are given in the RDB interface file.*
>
> *Example (will react on entities of category PLAYER and type CAR only):*
>
> > *`<Filter rdbCategory="RDB_OBJECT_CATEGORY_PLAYER" rdbType="RDB_OBJECT_TYPE_PLAYER_CAR"/>`*
>
> *Omitting the rdbType will add a filter for all objects of the given category*
>
> *Example (will react on all entities of category PLAYER):*
>
> > *`<Filter rdbCategory="RDB_OBJECT_CATEGORY_PLAYER"/>`*

OpenDRIVE objects

> *Objects defined in the OpenDRIVE file may be of designated types which are not part of the RDB specification. Therefore, the desired types of these objects may be provided as plain text as they also occur in the OpenDRIVE file.*
>
> *Example (will react on OpenDRIVE objects of type "road"):*
>
> > *`<Filter odrType="road" />`*

Note: individual objects may be excluded from the crash filter by explicitly specifying the attribute `excludeName` and providing the respective object's name string.

Within one `<Sensor>` definition, multiple Filters may be defined.

Example (will react on OpenDRIVE objects of type "road"):

```
<Sensor name="crash" type="video">
    <Load     lib="libModuleCrashSensor.so" path="" persistent="true" />
    <Output   scpUserMessage="true"><![CDATA[<Sound name="mySound"><Start id="0"/></Sound><Info><Message text="You crashed!" p
    <Filter   rdbCategory="RDB_OBJECT_CATEGORY_PLAYER" rdbType="RDB_OBJECT_TYPE_PLAYER_CAR"/>
    <Filter   rdbCategory="RDB_OBJECT_CATEGORY_PLAYER" rdbType="RDB_OBJECT_TYPE_PLAYER_TRUCK"/>
    <Filter   odrType="road" />
    <Filter   excludeName="myObject.flt"/>
    <Player   default="true" />
    <Debug    enable="false"/>
</Sensor>
```

Single Ray Sensor

availability: VTD 1.4.x+

packaging: in base package

This is a specific sensor which computes just a single ray from the sensor's mount point into the sensor direction. It computes the intersection of this ray and the bounding box of other objects (vehicles); it returns the position of these objects in sensor co-ordinates with the origin being at the sensor's mounting point. The sensor parameters (except for position, direction and range may not be modified.

```
<Sensor name="simpleSensor" type="radar">
    <Load     lib="libModuleSingleRaySensor.so" path="" persistent="true" />
    <Frustum  near="0.0" far="50.0" left="1.0" right="1.0" bottom="1.0" top="1.0" />
    <Cull     maxObjects="5" enable="true" />
    <Port     name="RDBout" number="48195" type="UDP" sendEgo="true" />
    <Player   default="true" />
    <Position dx="3.5" dy="0.0" dz="0.5" dhDeg="0.0" dpDeg="0.0" drDeg="0.0" />
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Filter   objectType="trafficSign"/>
    <Debug    enable="false"/>
</Sensor>
```

Multi-Ray Sensor

availability: VTD 2.0.2+

packaging: separate

The multi-ray sensor provides a means to interact with the **actual geometry** of the 3d database from within the moduleManager. For this purpose, there is a communication channel between MM and the imageGenerator. This means that you may also take continuous features of the environment (e.g. hills, road surface) into account for the sensing.

How it works

The sensor cone is split into `u * v` cells. In the center of each cell, a ray will be cast into the respective direction. The distances reported by each ray will be read and the minimum of all distances will be reported by the sensor as an output message of typ `RDB_PKG_ID_SENSOR_OBJECT_t`. The sensor will report no other information about the object etc. So, for now, it pretty much resembles what you would expect from a simplified model of an ultrasonic sensor.

Configuration

The sensor is configured quite like a perfect sensor plug-in. The only differences are:

- name of the plugin (`libModuleMultiRaySensor.so`)
- configuration of ray resolution (see `<Config...>` below

Example:

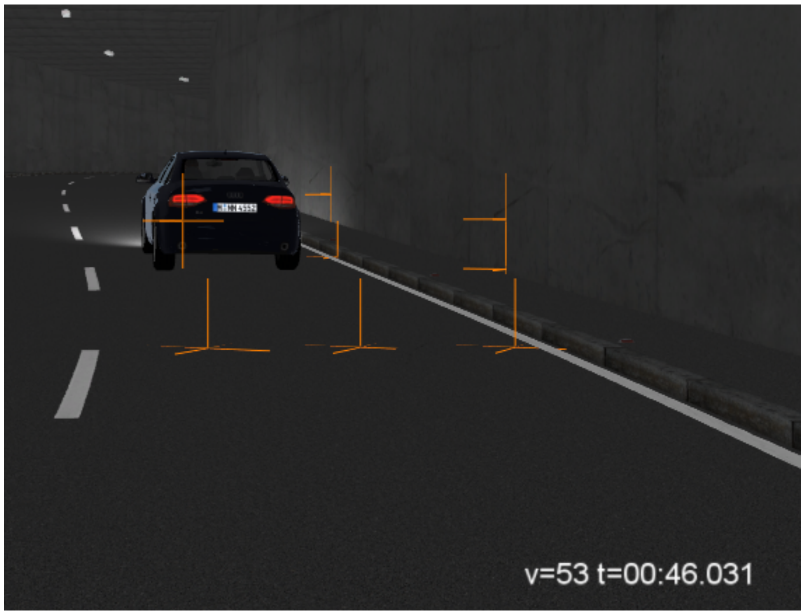
```
<Sensor name="multiRay" type="video">
    <Load     lib="libModuleMultiRaySensor.so" path="" persistent="true" />
    <Frustum  near="0.0" far="50.0" left="10.0" right="10.0" bottom="3.0" top="3.0" />
    <Config   noRaysHorizontal="3" noRaysVertical="3" verbose="false" />
    <Port     name="RDBout" number="48195" type="TCP" sendEgo="false" />
    <Player   default="true" />
    <Position dx="3.5" dy="0.0" dz="0.5" dhDeg="0.0" dpDeg="0.0" drDeg="0.0" />
    <Debug    enable="false" />
</Sensor>
```

If you wish to see the hit points in the imageGenerator (see images below), you will have to turn on the debug option *rayHits* in the TaskControl.

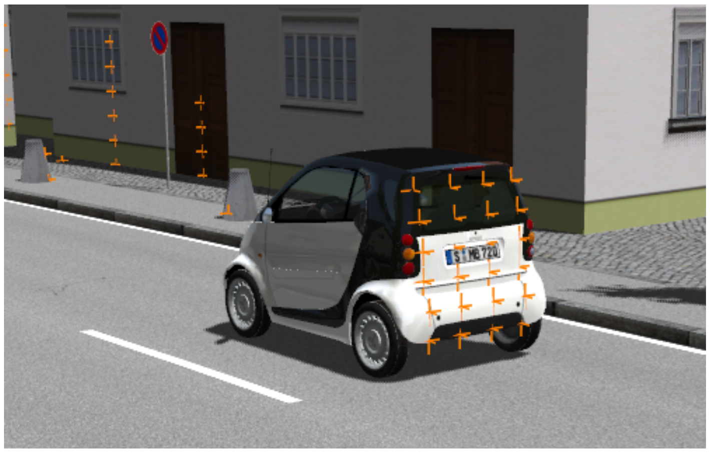
```
<TaskControl>
  :
  <Debug ...
    rayHits="true"/>
</TaskControl>
```

Result

When visualized, the hit points illustrate quite well the way the sensor works:



With a higher number of rays, results look a bit more illustrative:



Example

A setup including the multi-ray sensor is provided here:

vtd.2.0.3.addOns.MultiRaySensor.20170604.tgz

Filtering of Object Types

Overview

For the sensor plug-ins (exception: CrashSensor, see above), you may enable the detection of various object types using the `<Filter>` tag. Per default (i.e. if no filter is given), a sensor will detect all types of objects. So, once you start doing the filtering, make sure you add the filters for all object types that you wish to detect.

Example:

```
<Sensor name="camSensor" type="video">
    :
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Filter   objectType="trafficSign"/>
    <Filter   objectType="light"/>
    :
</Sensor>
```

Available filter types are:

```
- all
- vehicle
- pedestrian
- light
- trafficSign
- obstacle
- laneInfo (new in VTD 1.2.2)
- roadMarks (new in VTD 1.2.2)
```

Traffic Signs

For **traffic signs**, the filtering can also be limited to certain types, e.g. with the settings

```
<Filter   objectType="trafficSign" signType="274"/>
<Filter   objectType="trafficSign" signType="281"/>
```

the plugin will deliver only traffic signs of with OpenDRIVE numbers 247 and 281.

Per default, traffic signs will be filtered by the sensor plug-in to match the following types:

```
205
206
274
276
277
278
280
281
282
301
306
307
```

```
310
311
330
334
> 1000000
```

You may turn off the filtering of traffic signs completely by providing the following configuration line:

```
<Filter objectType="trafficSign" signType="-1"/>
```

Road Marks

Note: background information about road marks may also be found here: How do I read the roadmark information

For road marks, additional filter parameters are available which trigger the tesselation (default is OFF), the preview distance (default is 100m), and other details.

Example:

```
<Filter objectType="roadMarks" roadmarkPreviewDistance="100.0"
        tesselate="true" tesselateNoPoints= "10" tesselateFixedStep="true"
        tesselateOffsetZ="0.01" />
```

with:

- *tesselateNoPoints*: number of tesselation points used for discretization (default: 64)
- *tesselateFixedStep*: perform tesselation in fixed step width, i.e. variable number of resulting tesselation points (default: false, i.e. variable step width with fixed number of tesselation points)
- *roadmarkPreviewDistance*: maximum preview distance for road mark detection
- *tesselateOffsetZ*: distance between road surface and roadmark's upper edges; the default value is 0.02m and should not be altered unless you are sure that the database has been generated using a different z-offset for the road marks (see next paragraph).

Correlation between image data and road mark sensor:

In the 3d database road marks are generated as polygons with an offset from the underlying road surface. Therefore, your image processing algorithms may report them at slightly different places than what is reported from the (perfect) sensor. You may, however, adapt both representations and force consistent behavior by applying the following adaptations:

- Road Designer ROD
  - open the file `TT_SETUP.DAT` that applies to your project
  - set the variable `TED_ROADMARK_OFFSET` to the applicable value (default: 0.02m)
  - Example: `TED_ROADMARK_OFFSET 0.01`
- Sensor Plug-in
  - open the ModuleManager configuration file `moduleManager.xml`
  - provide a roadmark tesselation definition and set the attribute `tesselateOffsetZ` to the same value
  - Example: see above

Legacy Information:
When computing lane information and road marks within the sensor, make sure these data are not computed within the taskControl (standard until VTD 1.2.2). In order to disable the calculation within TC, change the file *Data/Setups/Current/Config/TaskControl/taskControl.xml* as follows:

```
<TaskControl>
:
<RoadInfo enable="false" .../>
:
</TaskControl>
```

Basic information (e.g. road position for players, road state) and contact points will be computed by the taskControl despite these settings.

JitterSensor

The *JitterSensor* may be used for deteriorating the accuracy of information retrieved by the perfect sensor. It will affect **all** `OBJECT_STATE` data after these have been calculated by the perfect sensor and before sending them via the `RDBout` channel. The position and speed vectors which are sent may be affected per component. On each component, a sinusoidal noise will be added to the actual signal. The user may specify *frequency* and *amplitude* of the noise per channel.

In addition, the user may also specify a periodic complete loss of signal for a given duration each time.

availability: VTD 2.0.x+

base class: perfect sensor

plugin: libModuleJitterSensor.so

parameterization:

The parameterization if performed in the ModuleManager's configuration file or via SCP.

Example (noise only):

```
<Sensor>
  :
  <Noise axis="x" amplitude="1.0" frequency="40.0"/>
  <Noise axis="p" amplitude="0.2" frequency="20.0"/>
  :
</Sensor>
```

For the attribute *axis*, you may provide

- x, xDot
- y, yDot
- z, zDot
- h, hDot
- p, pDot
- r, rDot

The *amplitude* is given in `[m]` or `[rad]`, respectively, the *frequency* is given in `[Hz]`

Example (noise and data loss):

```
<Sensor>
  :
  <Noise axis="x" amplitude="1.0" frequency="40.0"/>
  <Noise axis="p" amplitude="0.2" frequency="20.0"/>
  <DataLoss duration="0.2" frequency="0.5"/>
  :
</Sensor>
```

The periodic data loss in the example will be for 0.2s each time with a frequency of 0.5Hz.

Extended Roadmark Sensor

The *RoadmarkCnCSensor* may be used for calculating sophisticated roadmark information based on the position of roadmarks within the frustum of the sensor.

availability: VTD 2.0.2+

base class: perfect sensor

plugin: libModuleRoadmarkCnCSensor.so

description:

The difference between the "standard" and this "sophisticated" sensor is illustrated in the following table:

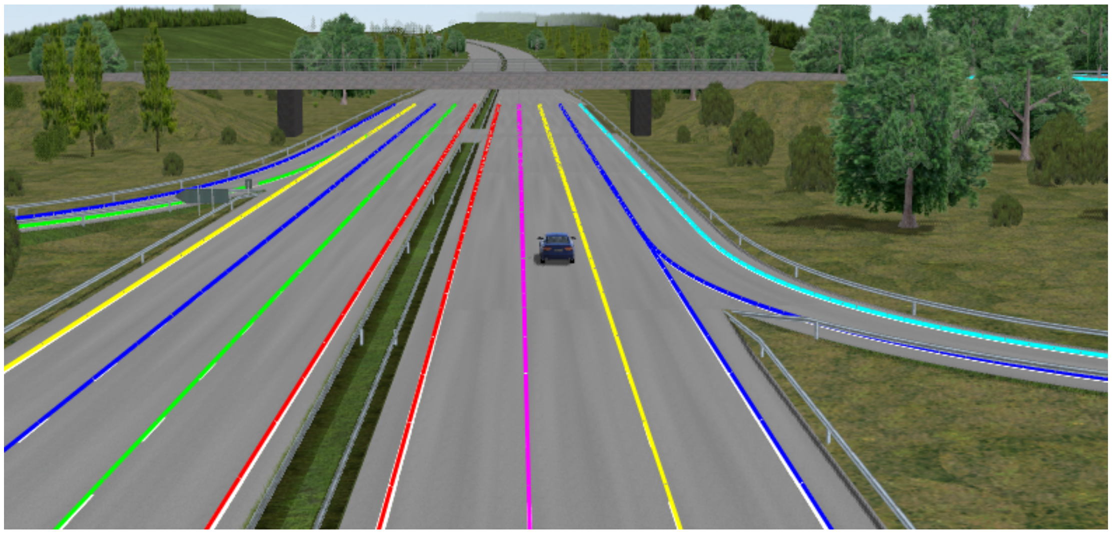
	PerfectSensor	**Sophisticated Sensor**
plug-in	libModulePerfectSensor.so	libModuleRoadmarkCnCSensor.so
junctions	partially supported	fully supported
tesselation	yes	yes
curvature information	yes	no
max. number of fragments	2	unlimited
successor/predecessor	yes	no
grouping	no	yes
search strategy	logic links to own position	global

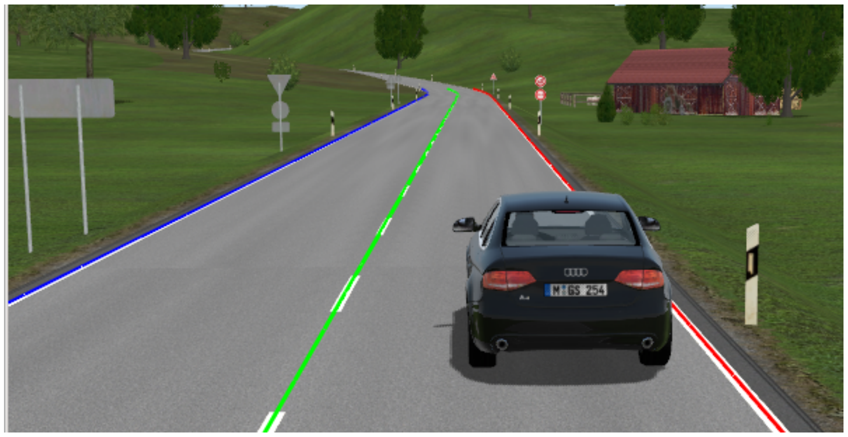
In contrast to the "standard" roadmark detection, this sensor will search globally in the OpenDRIVE database for all roadmarks and register the ones which are intersecting the sensor's frustum. Complex road situations may be illustrated in a much better and more complete way than with the standard sensor. The sensor may only be used for generating **tesselated** roadmark information.

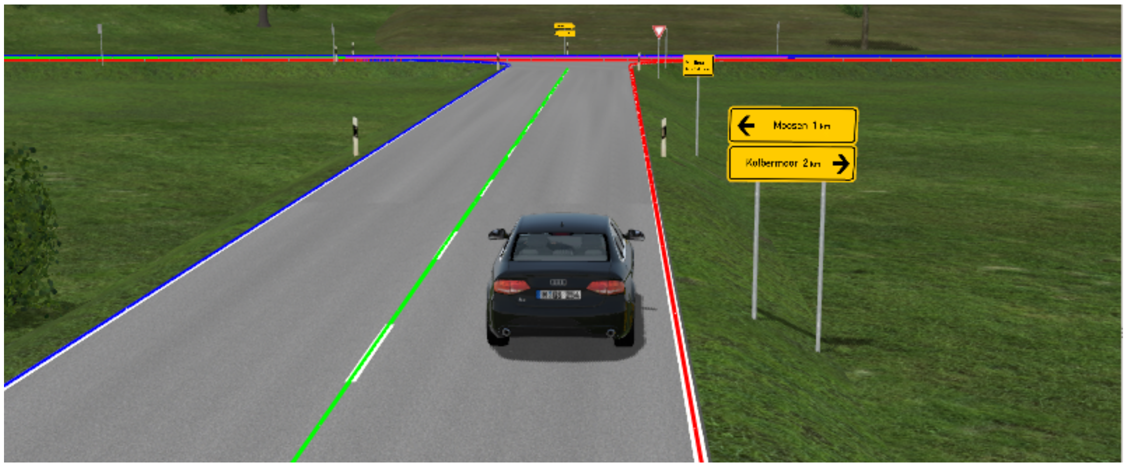
interpretation of resulting data:

Individual roadmark entries belonging to each other will be tagged with the same unique numeric id. This is to be found in the member *nextId* of the structure `RDB_ROADMARK_t`. The member *prevId* contains the number of the lane section where the roadmark originated. All members usually providing *curvature* information are **invalid**.

In the following pictures, roadmarks with the same unique id are colored identically (note: colors are repeated every five roadmarks).







parameterization:

The parameterization is performed in the ModuleManager's configuration file or via SCP.

Example:

```
<Sensor name="perfect" type="video">
    <Load     lib="libModuleRoadmarkCnCSensor.so" path="" persistent="true" />
    <Frustum  near="0.0" far="150" left="50" right="50" bottom="5.0" top="5.0" />
    <Cull     maxObjects="5" enable="true" />
    <Port     name="RDBout" number="48198" type="TCP" sendEgo="true" />
    <Player   default="true" />
    <Position dx="0.0" dy="0.0" dz="0.0" dhDeg="0.0" dpDeg="0.0" drDeg="0.0" />
    <Origin   type="inertial"/>
    <Filter   objectType="roadMarks" tesselate="true" tesselateNoPoints= "20" tesselateFixedStep="false" roadmarkPreviewDistan
    <Debug    enable="false" />
</Sensor>
```

The most important entries and attributes are:

- *Frustum*:
  - make sure the frustum is large (wide) enough to cover what you want to see
- *Position*:
  - a position near the ground is preferred
- *Filter*:
  - *fullScope*: this **must** be true
  - select the tesselation parameters at own discretion

Sensor Fusion

The output data of various sensors may be combined by a "fusion sensor". In order to use this functionality, define your original sensors as usual but **without** RDB output and then define a fusion sensor (also based on the perfect sensor) which collects the data and writes it to the RDB output. This principle is best explained by an example configuration of the ModuleManager:

```
<Sensor name="perfect" type="video">
    <Load     lib="libModulePerfectSensor.so" path="" persistent="true" />
    <Frustum  near="0.0" far="50.0" left="10.0" right="10.0" bottom="3.0" top="3.0" />
    <Cull     maxObjects="5" enable="true" />
    <Player   default="true" />
    <Position dx="3.5" dy="0.0" dz="0.5" dhDeg="0.0" dpDeg="0.0" drDeg="0.0" />
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Debug    enable="false" />
</Sensor>

<Sensor name="perfect2" type="video">
    <Load     lib="libModulePerfectSensor.so" path="" persistent="true" />
    <Frustum  near="0.0" far="50.0" left="40.0" right="10.0" bottom="3.0" top="3.0" />
    <Cull     maxObjects="5" enable="true" />
    <Player   default="true" />
    <Position dx="3.5" dy="0.0" dz="0.5" dhDeg="-90.0" dpDeg="0.0" drDeg="0.0" />
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Debug    enable="false"/>
</Sensor>

<Sensor name="collect" type="video">
    <Load     lib="libModulePerfectSensor.so" path="" persistent="true" />
    <Cull     maxObjects="0" enable="true" />
    <Player   default="true" />
    <Source   name="perfect" />
    <Source   name="perfect2" />
    <Port     name="RDBout" number="48195" type="UDP" sendEgo="false" />
    <Debug    enable="false" />
</Sensor>
```

Here, the first two sensors *perfect* and *perfect2* act as input for *collect*. All unique objects will be combined and transmitted by the latter sensor.

Dynamics Plug-ins provided with VTD

VTD comes with two built-in vehicle dynamics models - the single-track model (simple and fast) and the 5-mass-model (more complex, a bit slower). Both are described in this chapter. In order to select the one for your application, we recommend to proceed as follows:

- create a directory *Data/Setups/Current/Config/ModuleManager*
- copy the file *Data/Distros/Current/Config/ModuleManager/moduleManager.xml* to this directory
- open the file and activate the corresponding plug-in
  - for single-track dynamics activate the plugin "viTrafficDyn"
  - for 5-mass-model activate the plugin "viTrafficDynComplex"
- remove the other plug-in entry
- in order to work seamless, make sure the tag `<Player>` says `<Player default="true"/>` and does note have the attribute `name` defined.

For the detailed configuration of a vehicle's dynamics itself, please see also Vehicle Dynamics in the ModuleManager

Single-Track Model "TrafficDyn"

Overview

The *TrafficDyn* is provided by means of the plugin `libModuleTrafficDyn.so`. It is based on the same single-track dynamics model that is used also for all internal vehicles of the VTD traffic simulation module.You may configure multiple instances of the dynamics plugin in the ModuleManager's configuration file. Make sure that each plugin in the file has a unique name and is assigned to a dedicated vehicle. The following example shows the default definition of a trafficDyn plugin:

```
<DynamicsPlugin name="viTrafficDyn">
    <Load     lib="libModuleTrafficDyn.so" path=""/>
    <Player   default="true" />
    <Driver   ctrlLatLong="ghostdriver" />
    <Debug    enable="false"/>
```

```
</DynamicsPlugin>
```

Customization / Additional Funtionality

You may adapt the configuration of the trafficDyn with the following additional tags (see also SCP documentation):

- <Output>
- <Vehicle>

Sinusoidal Body Motion

For test purposes you may add sinusoidal offsets to the resulting x/y/z/h/p/r data of the vehicle dynamics before it is sent to the taskControl. Note: this only influences the data sent to TC, not the actual, internally held positions. The axis, amplitude and frequency may be specified individually using one `<Vehicle>` entry per definition.

Example:

```
<DynamicsPlugin name="viTrafficDyn">
    :
    <Vehicle  bodyAnimationAxis="0" bodyAnimationFrequency="0.5" bodyAnimationAmplitude="2.0" />
    <Vehicle  bodyAnimationAxis="1" bodyAnimationFrequency="0.5" bodyAnimationAmplitude="2.0" />
    :
</DynamicsPlugin>
```

The conventions are as follows:

- axis=0: x
- axis=1: y
- axis=2: z
- axis=3: h
- axis=4: p
- axis=5: r

The *frequency* is given in Hz, the *amplitude* in [m] and [rad], respectively

Complex (5-mass) Vehicle Dynamics

The 5-mass vehicle dynamics provides a more realistic behavior of the vehicle in contact with the road. It uses four contact points, so that rough roads, obstacles (e.g. speed bumps) etc. can interact better with the vehicle. The resulting motion of the vehicle body will also showw a more realistic behavior.

For activating this dynamics plug-in just make sure that the following entry is the active one in your configuration file `moduleManager.xml`

```
<DynamicsPlugin name="viTrafficDynComplex">
    <Load     lib="libModuleTrafficDynComplex.so" path=""/>
    <Player   default="true" />
    <Driver   ctrlLatLong="ghostdriver" />
    <Debug    enable="false"/>
</DynamicsPlugin>
```

That's basically it.

Running Multiple Instances

If you want to compute more than one vehicle using a dynamics plug-in, you have to do the following:

- in your scenario: create one *external* player per entity that you want to have computed by a plug-in
- instantiate one *uniquely named* plugin per vehicle and assign it *by name* to the given vehicle
- make sure there is no instance configured which grabs the *default* player
- place your modified MM configuration file `moduleManager.xml` either in `Data/Setups/Current/Config/ModuleManager` or in `Data/Projects/Current/Config/ModuleManager` (the latter one will supersede the former one)

Example:

- configuration with two vehicles named *A* and *B*
- Module Manager configuration file (@moduleManager.xml):

```
<!-- ##################################################################
     #
     # configuration file for modules 06.05.2017 by M. Dupuis
     #
     # (c)2017 by VIRES Simulationstechnologie GmbH
     #
     ##################################################################
-->
<RDB>
    <Port name="RDBraw" number="48190" type="TCP" />
</RDB>

<Debug    enable="false" />

<DynamicsPlugin name="viTrafficDynA">
    <Load     lib="libModuleTrafficDyn.so" path=""/>
    <Player   name="A" />
    <Debug    enable="false"/>
```

```
</DynamicsPlugin>

<DynamicsPlugin name="viTrafficDynB">
    <Load     lib="libModuleTrafficDyn.so" path=""/>
    <Player   name="B" />
    <Debug    enable="false"/>
</DynamicsPlugin>
```

Note: the assignment of the players to the plugin instance has to be *explicit* (i.e. `<Player name="A"/>` and `<Player name="B"/>` instead of `<Player default="true"/>`).

Light Source Test

Overview

This plug-in is not really a dynamics plug-in but it was based on the DynamicPlugin base class.

availability: on request only

purpose: illustrate online manipulation of extended light sources via RDB

description: the plugin attaches two light sources to the vehicle with player ID no. 1 and modifies (based on module manager frame counts) various properties of these light sources; information of what is being changed is shown as text on the IG screen.

notes: you may also use this example for directly composing and sending the data from your own application via RDB. In this case, just connect to the TC's RDB port and replace in the example below the call of the method `ModulePlugin::sendFeedback( myHandler.getMsg(), myHandler.getMsgTotalSize() );` with your own send routine. The same applies to the SCP commands (`sendSCPMessage()`) but these are not essential for the functionality of the light source control

ModuleManager Configuration

For activation, add the plug-in to your ModuleManager configuration:

```
<DynamicsPlugin name="lsTest">
    <Load     lib="libModuleLightSourceTest.so" path=""/>
    <Player   default="true" />
    <Debug    enable="false"/>
</DynamicsPlugin>
```

Code Excerpt

The following code fragment illustrates the key programming principles of light source control.

```
int
LightSourceTest::update( const unsigned long & frameNo , DynamicsIface* ifaceData )
{
    Framework::RDBHandler myHandler;

    if ( !ifaceData )
        return 0;

    if ( mOwnPlayerId < 0 )
        return 0;
    bool                sendComplex = true;
    static unsigned int count       = 0;
    static int          sLastState  = 5000;

    if ( sendComplex )
    {
        RDB_LIGHT_SOURCE_t* pInsData = ( RDB_LIGHT_SOURCE_t* ) myHandler.addPackage( ifaceData->mSimTime, ifaceData->mFrameNo,

        if ( !pInsData )
            return 0;

        count += 5;

        count = count % 15000;

        double simTimeFac = 2.0;

        if ( count > 13000 )
        {
            pInsData->base.id            = 35;
            pInsData->base.pos.type      = RDB_COORD_TYPE_PLAYER;
            pInsData->base.playerId      = 1;
            pInsData->base.templateId    = -1;
            pInsData->base.state         = 1;

            pInsData = ( RDB_LIGHT_SOURCE_t* ) myHandler.addPackage( ifaceData->mSimTime, ifaceData->mFrameNo, RDB_PKG_ID_LIGH
            pInsData->base.id            = 1;
            pInsData->base.pos.type      = RDB_COORD_TYPE_PLAYER;
            pInsData->base.playerId      = 1;
            pInsData->base.templateId    = -1;
            pInsData->base.state         = 2;

            if ( sLastState != 13 )
            {
                sendSCPMessage( "<Symbol name=\"expl01\" > <Text data=\"all light sources off\" colorRGB=\"0x00ff00\" size=\"3
                sLastState = 13;
            }
        }
        else if ( count > 11000 )
        {
            pInsData->base.id            = 35;
            pInsData->base.pos.type      = RDB_COORD_TYPE_PLAYER;
            pInsData->base.playerId      = 1;
            pInsData->base.templateId    = 0;
            pInsData->base.state         = 1;
            pInsData->base.pos.x         = 1.50;
            pInsData->base.pos.y         = 0.00;
            pInsData->base.pos.z         = 0.30;
            pInsData->base.pos.h         = 0.0;
            pInsData->base.pos.p         = 0.0;
            pInsData->base.pos.r         = 0.00;
            pInsData->ext.nearFar[0]     =  1.0;
            pInsData->ext.nearFar[1]     = 50.0;
            pInsData->ext.frustumLRBT[0] = -1.7f;
            pInsData->ext.frustumLRBT[1] =  1.7f;
            pInsData->ext.frustumLRBT[2] = -1.5f;
            pInsData->ext.frustumLRBT[3] =  1.5f;
            pInsData->ext.intensity[0]   = 1.0;
            pInsData->ext.intensity[1]   = 1.0;
            pInsData->ext.intensity[2]   = 1.0;
            pInsData->ext.atten[0]       = 0.3;
            pInsData->ext.atten[1]       = 0.02;
            pInsData->ext.atten[2]       = 0.0005;

            pInsData = ( RDB_LIGHT_SOURCE_t* ) myHandler.addPackage( ifaceData->mSimTime, ifaceData->mFrameNo, RDB_PKG_ID_LIGH
            pInsData->base.id            = 1;
            pInsData->base.pos.type      = RDB_COORD_TYPE_PLAYER;
            pInsData->base.playerId      = 1;
            pInsData->base.templateId    = 0;
            pInsData->base.state         = 2;
            pInsData->base.pos.x         = 1.50;
            pInsData->base.pos.y         = 1.00;
            pInsData->base.pos.z         = 0.30;
            pInsData->base.pos.h         = 0.0;
            pInsData->base.pos.p         = 0.0;
            pInsData->base.pos.r         = 0.00;

            if ( sLastState != 11 )
            {
                sendSCPMessage( "<Symbol name=\"expl01\" > <Text data=\"mixing complex and simple\" colorRGB=\"0x00ff00\" size
                sLastState = 11;
            }
        }
        else if ( count > 9000 )
        {
            pInsData->base.id            = 35;
            pInsData->base.pos.type      = RDB_COORD_TYPE_PLAYER;
            pInsData->base.playerId      = 1;
            pInsData->base.templateId    = 0;
            pInsData->base.state         = 1;
            pInsData->base.pos.x         = 1.50;
            pInsData->base.pos.y         = 0.00;
            pInsData->base.pos.z         = 0.30;
            pInsData->base.pos.h         = 0.0;
            pInsData->base.pos.p         = 0.0;
            pInsData->base.pos.r         = 0.00;
            pInsData->ext.nearFar[0]     =  1.0;
            pInsData->ext.nearFar[1]     = 50.0;
            pInsData->ext.frustumLRBT[0] = -1.7f;
            pInsData->ext.frustumLRBT[1] =  1.7f;
            pInsData->ext.frustumLRBT[2] = -1.5f;
            pInsData->ext.frustumLRBT[3] =  1.5f;
            pInsData->ext.intensity[0]   = 1.0;
            pInsData->ext.intensity[1]   = 1.0;
            pInsData->ext.intensity[2]   = 1.0;
            pInsData->ext.atten[0]       = 0.3;
            pInsData->ext.atten[1]       = 0.02;
            pInsData->ext.atten[2]       = 0.0005;

            pInsData = ( RDB_LIGHT_SOURCE_t* ) myHandler.addPackage( ifaceData->mSimTime, ifaceData->mFrameNo, RDB_PKG_ID_LIGH
            pInsData->base.id            = 35;
            pInsData->base.pos.type      = RDB_COORD_TYPE_PLAYER;
            pInsData->base.playerId      = 1;
            pInsData->base.templateId    = 0;
            pInsData->base.state         = 1;
            pInsData->base.pos.x         = 0.5 + 0.01 * sin( ifaceData->mSimTime );
            pInsData->base.pos.y         = 0.5 + 0.01 * sin( ifaceData->mSimTime );
            pInsData->base.flags        |= RDB_LIGHT_SOURCE_FLAG_STENCIL;

            if ( sLastState != 9 )
            {
                sendSCPMessage( "<Symbol name=\"expl01\" > <Text data=\"stencil mask test\" colorRGB=\"0x00ff00\" size=\"30.0\
                sLastState = 9;
            }
        }
        else if ( count > 7000 )
        {
            pInsData->base.id            = 35;
            pInsData->base.pos.type      = RDB_COORD_TYPE_PLAYER;
            pInsData->base.playerId      = 1;
            pInsData->base.templateId    = 0;
            pInsData->base.state         = 1;
            pInsData->base.pos.x         = 1.50;
            pInsData->base.pos.y         = 0.00;
            pInsData->base.pos.z         = 0.30;
            pInsData->base.pos.h         = 0.0;//0.1 * sin( 0.1 * mSimTime );
            pInsData->base.pos.p         = 0.0;//0.05 * sin( mSimTime );
            pInsData->base.pos.r         = 0.00;
            pInsData->ext.nearFar[0]     =  1.0;
            pInsData->ext.nearFar[1]     = 50.0;
            pInsData->ext.frustumLRBT[0] = -1.7f;// + 0.1f * sin( mSimTime );
            pInsData->ext.frustumLRBT[1] =  1.7f;// + 0.1f * sin( mSimTime );
            pInsData->ext.frustumLRBT[2] = -1.5f;// + 0.1f * sin( mSimTime );
            pInsData->ext.frustumLRBT[3] =  1.5f;// + 0.1f * sin( mSimTime );
            pInsData->ext.intensity[0]   = 0.5 * ( 1.0 + sin( simTimeFac * ifaceData->mSimTime ) );
            pInsData->ext.intensity[1]   = 0.5 * ( 1.0 + sin( simTimeFac * ifaceData->mSimTime ) );
            pInsData->ext.intensity[2]   = 0.5 * ( 1.0 + sin( simTimeFac * ifaceData->mSimTime ) );
            pInsData->ext.atten[0]       = 0.3;
            pInsData->ext.atten[1]       = 0.02;
            pInsData->ext.atten[2]       = 0.0005;

            if ( sLastState != 7 )
            {
                sendSCPMessage( "<Symbol name=\"expl01\" > <Text data=\"modifying intensity\" colorRGB=\"0x00ff00\" size=\"30.
                sLastState = 7;
            }
        }
        else if ( count > 5000 )
        {
            pInsData->base.id             = 35;
            pInsData->base.pos.type       = RDB_COORD_TYPE_PLAYER;
            pInsData->base.playerId       = 1;
            pInsData->base.templateId     = 0;
            pInsData->base.state          = 1;
            pInsData->base.pos.x          = 1.50;
            pInsData->base.pos.y          = 0.00;
            pInsData->base.pos.z          = 0.30;
            pInsData->base.pos.h          = 0.0;//0.1 * sin( 0.1 * mSimTime );
            pInsData->base.pos.p          = 0.0;//0.05 * sin( mSimTime );
            pInsData->base.pos.r          = 0.00;
            pInsData->ext.nearFar[0]      =    1.0 + ( 0.2 + sin( simTimeFac * mSimTime ) );
            pInsData->ext.nearFar[1]      =  100.0 + 1.0 * sin( simTimeFac * mSimTime );
            pInsData->ext.frustumLRBT[0]  = -1.7f;
            pInsData->ext.frustumLRBT[1]  =  1.7f;
            pInsData->ext.frustumLRBT[2]  = -1.5f;
            pInsData->ext.frustumLRBT[3]  =  1.5f;
            pInsData->ext.intensity[0]    = 1.0;
            pInsData->ext.intensity[1]    = 1.0;
            pInsData->ext.intensity[2]    = 1.0;
            pInsData->ext.atten[0]        = 0.3;
            pInsData->ext.atten[1]        = 0.02;
            pInsData->ext.atten[2]        = 0.0005;

            if ( sLastState != 5 )
            {
                sendSCPMessage( "<Symbol name=\"expl01\" > <Text data=\"modifying frustum near/far\" colorRGB=\"0x00ff00\" size
                sLastState = 5;
            }
        }
        else if ( count > 3000 )
        {
            pInsData->base.id            = 35;
            pInsData->base.pos.type      = RDB_COORD_TYPE_PLAYER;
            pInsData->base.playerId      = 1;
            pInsData->base.templateId    = 0;
            pInsData->base.state         = 1;
            pInsData->base.pos.x         = 1.50;
            pInsData->base.pos.y         = 0.00;
            pInsData->base.pos.z         = 0.30;
            pInsData->base.pos.h         = 0.0;//0.1 * sin( 0.1 * mSimTime );
            pInsData->base.pos.p         = 0.0;//0.05 * sin( mSimTime );
            pInsData->base.pos.r         = 0.00;
            pInsData->ext.nearFar[0]     =   1.0;
            pInsData->ext.nearFar[1]     =  100.0;
            pInsData->ext.frustumLRBT[0] = -1.7f - 1.0f * sin( simTimeFac * mSimTime );
            pInsData->ext.frustumLRBT[1] =  1.7f + 1.0f * sin( simTimeFac * mSimTime );
            pInsData->ext.frustumLRBT[2] = -1.5f; // + 1.0f * sin( simTimeFac * mSimTime );
            pInsData->ext.frustumLRBT[3] =  1.5f; // + 1.0f * sin( simTimeFac * mSimTime );
            pInsData->ext.intensity[0]   = 1.0;
            pInsData->ext.intensity[1]   = 1.0;
            pInsData->ext.intensity[2]   = 1.0;
            pInsData->ext.atten[0]       = 0.3;
            pInsData->ext.atten[1]       = 0.02;
            pInsData->ext.atten[2]       = 0.0005;

            //fprintf( stderr, "LightSourceTest::update: frustum n/f/l/r/b/t = %.3f / %.3f / %.3f / %.3f / %.3f / %.3f\n",
            //                                pInsData->ext.nearFar[0], pInsData->ext.nearFar[1], pInsData->ext.frustumLRBT[0],
            //                                pInsData->ext.frustumLRBT[1], pInsData->ext.frustumLRBT[2], pInsData->ext.frustumLI

            if ( sLastState != 3 )
            {
                sendSCPMessage( "<Symbol name=\"expl01\" > <Text data=\"modifying frustum left/right\" colorRGB=\"0x00ff00\" s
                sLastState = 3;
            }
        }

        else if ( count > 1000 )
        {
            pInsData->base.id            = 35;
            pInsData->base.pos.type      = RDB_COORD_TYPE_PLAYER;
            pInsData->base.playerId      = 1;
            pInsData->base.templateId    = 0;
            pInsData->base.state         = 1;
            pInsData->base.pos.x         = 1.50;
            pInsData->base.pos.y         = 0.00;
            pInsData->base.pos.z         = 0.30;
            pInsData->base.pos.h         = 0.4 * sin( simTimeFac * 0.5 * mSimTime );
            pInsData->base.pos.p         = 0.05 * sin( simTimeFac * mSimTime );
            pInsData->base.pos.r         = 0.00;
            pInsData->ext.nearFar[0]     =   1.0f;
            pInsData->ext.nearFar[1]     = 100.0f;
            pInsData->ext.frustumLRBT[0] =  -1.7f;
            pInsData->ext.frustumLRBT[1] =   1.7f;
            pInsData->ext.frustumLRBT[2] =  -1.5f;
            pInsData->ext.frustumLRBT[3] =   1.5f;
            pInsData->ext.intensity[0]   = 1.0;
            pInsData->ext.intensity[1]   = 1.0;
            pInsData->ext.intensity[2]   = 1.0;
            pInsData->ext.atten[0]       = 0.3;
            pInsData->ext.atten[1]       = 0.02;
            pInsData->ext.atten[2]       = 0.0005;

            //fprintf( stderr, "LightSourceTest::update: hdg / pitch = %.3f / %.3f\n",
            //                                pInsData->base.pos.h, pInsData->base.pos.p );

            if ( sLastState != 1 )
            {
                sendSCPMessage( "<Symbol name=\"expl01\" > <Text data=\"modifying heading/pitch\" colorRGB=\"0x00ff00\" size=\
                sLastState = 1;
            }
        }
        else
        {
            pInsData->base.id            = 35;
            pInsData->base.pos.type      = RDB_COORD_TYPE_PLAYER;
            pInsData->base.playerId      = 1;
            pInsData->base.templateId    = 0;
            pInsData->base.state         = ( int ) ( 2.0 * ( sin( simTimeFac * mSimTime ) + 1.0 ) );
            pInsData->base.pos.x         = 1.50;
            pInsData->base.pos.y         = 0.00;
            pInsData->base.pos.z         = 0.30;
            pInsData->base.pos.h         = 0.0;
            pInsData->base.pos.p         = 0.0;
            pInsData->base.pos.r         = 0.00;
            pInsData->ext.nearFar[0]     =   1.0f;
            pInsData->ext.nearFar[1]     = 100.0f;
            pInsData->ext.frustumLRBT[0] =  -1.7f;
            pInsData->ext.frustumLRBT[1] =   1.7f;
            pInsData->ext.frustumLRBT[2] =  -1.5f;
            pInsData->ext.frustumLRBT[3] =   1.5f;
            pInsData->ext.intensity[0]   = 1.0;
            pInsData->ext.intensity[1]   = 1.0;
            pInsData->ext.intensity[2]   = 1.0;
            pInsData->ext.atten[0]       = 0.3;
            pInsData->ext.atten[1]       = 0.02;
```

```
            pInsData->ext.atten[2]       = 0.0005;

            if ( sLastState != 0 )
            {
                sendSCPMessage( "<Symbol name=\"expl01\" > <Text data=\"modifying state\" colorRGB=\"0x00ff00\" size=\"30.0\" 
                sLastState = 0;
            }
        }

        myHandler.addPackage( ifaceData->mSimTime, ifaceData->mFrameNo, RDB_PKG_ID_END_OF_FRAME );

        ModulePlugin::sendFeedback( myHandler.getMsg(), myHandler.getMsgTotalSize() );
    }
    else
    {
        RDB_MSG_t* msg = 0;

        RDB_LIGHT_SOURCE_BASE_t* pInsData = ( RDB_LIGHT_SOURCE_BASE_t* ) myHandler.addPackage( ifaceData->mSimTime, ifaceData-

        if ( !pInsData || !msg )
            return 0;

        pInsData->id         = 35;
        pInsData->templateId = 0;
        pInsData->pos.type   = RDB_COORD_TYPE_PLAYER;
        pInsData->playerId   = mOwnPlayerId;
        pInsData->pos.x      = 1.50;
        pInsData->pos.y      = 0.00;
        pInsData->pos.z      = 0.30;
        pInsData->pos.h      = sin( ifaceData->mSimTime );
        pInsData->pos.p      = 0.00;
        pInsData->pos.r      = 0.00;
        pInsData->state      = 1;

        pInsData++;

        pInsData->id         = 22;
        pInsData->templateId = 1;
        pInsData->pos.type   = RDB_COORD_TYPE_PLAYER;
        pInsData->playerId   = mOwnPlayerId;
        pInsData->pos.x      = 1.50;
        pInsData->pos.y      = 0.00;
        pInsData->pos.z      = 0.30;
        pInsData->pos.h      = fmod( 3.0 * M_PI * ifaceData->mSimTime, 2.0 * M_PI );
        pInsData->pos.p      = 0.00;
        pInsData->pos.r      = 0.00;
        pInsData->state      = 6;

        myHandler.addPackage( ifaceData->mSimTime, ifaceData->mFrameNo, RDB_PKG_ID_END_OF_FRAME );

        ModulePlugin::sendFeedback( myHandler.getMsg(), myHandler.getMsgTotalSize() );
    }

    mLastFrameNo = mFrameNo;

    return 1;
}
```

Dynamics From File

Overview

This plug-in may be used for replaying data from previous recordings of time-stamped trajectories. The format supported by the plug-in is OpenSCENARIO (and an additional custom format).

file: `libModuleDynFromFile.so`

availability: on request only

purpose: merge players on recorded trajectories into the simulation

description: the plug-in reads a single record file. This file should contain time-stamped trajectories of any number of players whose trajectories shall not be computed by the VTD driver but replayed from the file itself. Players are created automatically by the TC upon the first replay message unless they are already present in the scenario. Where timestamps in the file do not match with the actual timestamps of the simulation, the plug-in will use linear interpolations for position and orientation values.

ModuleManager Configuration

For activation, add the plug-in to your ModuleManager configuration:

```
<DynamicsPlugin name="fileDyn">
    <Load       lib="libModuleDynFromFile.so" path=""/>
    <Player     default="false" />
    <Config     verbose="true"/>
    <File       name="IGLinks/Project/Recordings/3rdParty/Trajectory.xosc" format="XOSC" />
    <Output     sendMultiplePlayers="true"/>
    <Debug      enable="false"/>
</DynamicsPlugin>
```

Make sure you set the property `<Output sendMultiplePlayers="true"/>`. For the *verbose* mode, set whatever seems applicable to your case

Additional Information

An example for the record file is given here (note: there is still a flaw of the OSC specification included; this will be fixed in connection with the OSC spec.)

```
<?xml version="1.0" encoding="utf-8"?>
<OSCRouting>
    <fileHeader revMajor="0" revMinor="1" date="2016-02-26T10:00:00" description="Beispieltrajektorie fuer PEGASUS" author="An
    <observer name="" refId="42" type="">
        <general name="" closed="false"/>
        <waypoints>
            <waypoint timestamp="0.0">
                <position>
                    <positionWorld x="100" y="10" z="9.5" h="0" p="0" r="0"/>
                </position>
                <continuation>
                    <shape purpose="positioning">
                        <polyline/>
                    </shape>
                </continuation>
            </waypoint>
            <waypoint timestamp="10.5">
                <position>
                    <positionWorld x="101" y="10" z="9.5" h="0" p="0" r="0"/>
                </position>
                <continuation>
                    <shape purpose="positioning">
                        <polyline/>
                    </shape>
                </continuation>
            </waypoint>
            <waypoint timestamp="11.0">
                <position>
                    <positionWorld x="102" y="10" z="9.5" h="0" p="0" r="0"/>
                </position>
                <continuation>
                    <shape purpose="positioning">
                        <polyline/>
                    </shape>
                </continuation>
            </waypoint>
            <waypoint timestamp="11.5">
                <position>
                    <positionWorld x="103" y="10" z="9.5" h="0" p="0" r="0"/>
                </position>
                <continuation>
                    <shape purpose="positioning">
                        <polyline/>
                    </shape>
                </continuation>
            </waypoint>
            <waypoint timestamp="12.0">
                <position>
                    <positionWorld x="104" y="10" z="9.5" h="0" p="0" r="0"/>
                </position>
                <continuation>
                    <shape purpose="positioning">
                        <polyline/>
                    </shape>
                </continuation>
            </waypoint>
        </waypoints>
    </observer>
</OSCRouting>
```

note: the tag `<observer>` is actually not correct but consistent with the OSC specification; the `refId` will be interpreted as player's numeric ID.

Linking to SUMO

The link to SUMO is done via a special (sensor) plug-in of the module manager: `libModuleSUMOGateway.so`. It is mandatory that you have this plug-in in order to perform co-simulation between VTD and SUMO.

Operation Mode

In order to work properly, the moduleManager shall be operated in frame-synchronous mode. See also

Full-sync operation of ModuleManager

Configuration File

The configuration is performed "as usual" in the config file that is read by the ModuleManager.

Example:

```
<Sensor name="sumo" type="video">
    <Load   lib="libModuleSUMOGateway.so" path="" persistent="true" />
    <Port   name="SUMOout" number="8813" type="TCP" sendEgo="true" client="true" server="127.0.0.1" />
    <Port   name="RDBout" number="49175" type="TCP" sendEgo="false" />
    <SUMO   radiusIn="1500.0" radiusOut="1550.0" radiusFar="100000.0" verbose="false" showTrace="false" sendVehicleFrontEdge="
    <Player default="true" />
    <Config useSyncMask="true" />
    <Debug  enable="false"/>
</Sensor>
```

The following tags are to be taken care of:

`<Port>`

Apart from the RDB output port, the user also has to define the link port to SUMO. This is done via the port named `SUMOout`. Per default, it should be a TCP client port connecting to an instance of SUMO running on the same computer. Depending on your actual configuration, you will have to adapt these parameters.

`<SUMO>`

This is the actual tag describing the interaction between VTD and SUMO. Its attributes are:

- *radiusIn:*
  - radius around own vehicle where players which are closer than the given value will be imported from SUMO to VTD
  - type: float
  - values: any
- *radiusOut:*
  - radius around own vehicle where players which are farther than the given value will be returned from VTD to SUMO
  - type: float
  - values: any (larger than *radiusIn*)
- *radiusFar:*
  - radius around own vehicle where players from SUMO will be read and reported on RDBout but will not be sent to VTD itself
  - type: float
  - values: any (larger than *radiusOut*)
- *pathEndAction:*
  - action for vehicles which reach the end of their path while still being managed by VTD (i.e. outside SUMO)
  - values: `stop`, `continue`, `delete`
  - default: `stop`
- *verbose:*
  - enable / disable verbosity, i.e. debug output
  - values: `true`, `false`
  - default: `false`
- *showTrace:*
  - show trace of EGO vehicle in shell output (for debugging only)
  - values: `true`, `false`
  - default: `false`
- *startArgs:*
  - argument string for staring the SUMO application upon initialization of the simulation
  - values: valid file path
- *sendVehicleFrontEdge:*
  - SUMO is using the vehicle front edge as reference point, therefore, this attribute should be provided in the configuration and set to `true`
  - values: `true`, `false`
  - default: `false`
- *globalHdgOffsetDeg:*
  - With newer versions of SUMO, there may be issues concerning the orientation of vehicles in SUMO vs. VTD.
    In this case, the attribute *globalHdgOffsetDeg* should be set to **-90.0**
  - type: float
  - values: any
  - default: `0.0`
- *globalHdgScale:*
  - With newer versions of SUMO, there may be issues concerning the orientation of vehicles in SUMO vs. VTD.
    In this case, the attribute *globalHdgScale* should be set to **-1.0**
  - type: float
  - values: any
  - default: `1.0`

`<Config>`

This tag provides configuration settings for the interaction of the plug-in and VTD. Attributes are:

- *useSyncMask:*
  - register the plug-in to use the mask-based sync mechanism between TC and other external components; required if more than one external component is connected to VTD.
  - values: `true`, `false`
  - default: `false`
- *showTiming:*
  - show additional information in the console about processing times of the update routine and of the message handling routine of the plugin
  - values: `true`, `false`

- default: `false`
- *useOpenDriveLanes:*
    - Lane IDs have to be mapped between VTD and SUMO; for complex databases which include inner border lanes, it is useful to map the lane IDs using the actual OpenDRIVE lane IDs, not a default lane 0; the complete impact of this swwitch has still to be investigated; it originated from a debug session
    - type: bool
    - values: `true`, `false`
    - default: `false`
- *synchronizeInit:*
    - Large road networks with lots of vehicles may take some time until SUMO has loaded them and is ready-to-go. In ordert to prevent VTD from starting prematurely, this switch - if enabled - will make sure that SUMO is ready and has completely initialized before VTD is started.
    - type: bool
    - values: `true`, `false`
    - default: `false`
- *createExternalEntities:*
    - SUMO entities will, per default, be translated into *internally* animated VTD entities. By setting this attributue to `true`, VTD will instead create externally controlled entities (note: if you wish to use VTD drivers for them, make sure the external vehicle dynamics does also request driver commands from the traffic module)
    - values: `true`, `false`
    - default: `false`

SUMO Fixes

SUMO will complain about vehicles driving around in a database in a "free" fashion (i.e. vehicles for which no path can be identified in SUMO) and it will exit. In order to avoid premature exiting of the program, the corresponding `return` statement has been disabled in the file @MSBaseVehicle.cpp. **NOTE:** this is a workaround only and has to be properly implemented by the SUMO developers!

```
bool
MSBaseVehicle::hasValidRoute(std::string& msg) const {
    MSRouteIterator last = myRoute->end() - 1;
    // check connectivity, first
    for (MSRouteIterator e = myCurrEdge; e != last; ++e) {
        if ((*e)->allowedLanes(**(e + 1), myType->getVehicleClass()) == 0) {
            msg = "No connection between '" + (*e)->getID() + "' and '" + (*(e + 1))->getID() + "'.";
          //  return false;        // Marius Dupuis, Dec. 19, 2014!!!
        }
    }
    last = myRoute->end();
    // check usable lanes, then
    for (MSRouteIterator e = myCurrEdge; e != last; ++e) {
        if ((*e)->prohibits(this)) {
            msg = "Edge '" + (*e)->getID() + "' prohibits.";
            return false;
        }
    }
    return true;
}
```

Further Instructions

The following instructions are based on the assumption that your project be called *SUMO* or that you received the corresponding project from us.

Preparation of Vehicle Database

directory: `VTD.2.0/Data/Projects/SUMO/Scripts/SUMO-scenario/vtdVehicles`

steps:

- copy relevant vehicle `.xml` files from `Data/Distros/Current/Config/Players/Vehicles` to sub-dir `Vehicles`
- if applicable, assign specific vehicle types in file `translation.txt`
- run the script `python translateVehTypes.py > vtd_vehicle_types.add.xml`

Preparation of SUMO

- set VTD environment variable `SUMO_ROOT_DIR`
- convert your OpenDRIVE road network

```
netconvert --opendrive-files ../../UserDataVTD/Crossing8/Crossing8Course.xodr -o ../../UserDataVTD/Crossing8/Crossing8.net.xml
```

- generate your road trips / traffic

```
randomTrips.py -n ../../../../UserDataVTD/Crossing8/Crossing8.net.xml -e 20 -l -o ../../../../UserDataVTD/Crossing8/trips.trip

duarouter.exe -n ../../../UserDataVTD/Crossing8/Crossing8.net.xml -t ../../../UserDataVTD/Crossing8/trips.trips.xml -o ../../.

duarouter --repair -n ../../UserDataVTD/Crossing8/Crossing8.net.xml -r ../../UserDataVTD/Crossing8/Crossing8.rou.xml -o ../../
```

Starting SUMO

SUMO will typically be started by the script `VTD.2.0/Data/Projects/SUMO/Scripts/startSUMO.sh` which will be called by the ModuleManager plug-in `SUMOGateway`. For manual start or if the script is disabled, you may use the following command sequence:

```
sumo --step-length 1.000 -c ../../UserDataVTD/Crossing8/Crossing8.sumo.cfg -S -a ../../UserDataVTD/vtdVehicles
/vtd_vehicle_types.add.xml
sumo-gui.exe --step-length 1.000 -c ../../../UserDataVTD/Crossing8/Crossing8.sumo.cfg -S -a ../../../UserDataVTD/vtdVehicles
/vtd_vehicle_types.add.xml
```

Note: the file `vtd_vehicle_types.add.xml` must be used, so that SUMO and VTD will use the same vehicle database.

Custom ModuleManager Plug-ins

VTD provides a development environment for the creation of custom moduleManager plug-ins. First, make sure you have a license and the libraries for the moduleManager plug-in API. These are located at `Develop/Modules`

First step: compile the example

The source code of the PerfectSensor is provided as an example under `Develop/Modules/PerfectSensor`. This example may be compiled with the following steps:

1. Prerequisites:
    1. g++ 4.2.1 or higher
2. Go to `Develop/Modules/PerfectSensor`
3. Call @qmake
4. Call `make`
5. The plugin will be compiled and will be stored at `Develop/bin/Plugins`

Due to requirements for backward compatibility, the moduleManager and its plug-ins are currently made available for the indicated g++ version only. The development package may be made available for other g++ versions but this requires the entire tool-chain to be compiled in a different way.

Note: In case you are using a `c++11` environment, the compatibility with our base libraries which have been compiled with gcc4 might not be given. In this case, set the variable `_GLIBCXX_USE_CXX11_ABI` accordingly. See also https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_macros.html.

Understanding the API

The API is located in the sub-directory `Common`.

You may write sensor plug-ins, dynamics plug-ins or other moduleManager plug-ins specialized to your purpose.

The class hierarchy is as follows

```
ModulePlugin
    SensorPlugin
        PerfectSensor
    DynamicsPlugin
        SampleDyn
```

Therefore, if you develop your own plug-ins they should, preferably be derived from `SensorPlugin` or `DynamicsPlugin`. Only if both base classes don't fit your needs should you derive your own plug-in from the base class `ModulePlugin`.

Note: The class `ModulePlugin` provides the following members that allow you to identify the name and ID of e.g. a sensor's carrier:

```
    std::string mOwnPlayerName;
    int         mOwnPlayerId;
    bool        mOwnPlayerNameDefined;
```

The member `mOwnPlayerNameDefined` will be set to true once the other two members are filled with valid data.

Working with the data interfaces

Data which is received by the moduleManager via RDB (originating from the taskControl) is stored in so-called data interfaces. The same hierarchy applies as for the plug-ins:

```
ModuleIface
    SensorIface
    DynamicsIface
```

Each Plug-in has one input Iface and one output Iface. The input Iface represents the whole data received by the moduleManager's input routines (i.e. RDB) and the output Iface represents all data generated within the plug-in (e.g. copies of the incoming data that have been found to be relevant within a sensor).

The standard names for these interfaces are:

```
IN:
mModuleInData
    mSensorInData
    mDynamicsInData

OUT:
mModuleOutData
    mSensorOutData
    mDynamicsOutData
```

The data interfaces are based on the RDB structure, therefore working with them should be quite straightforward.
All data is stored in a map of vectors of RDB-compliant data types. Once you have a pointer to a valid interface,
you may extract data in the following way:

Example: parse object information and find the one about the ownship

```
    ModuleIface::DataVec*          objVec  = 0;
    ModuleIface::DataMap           dataMap = mModuleInData->getDataMap();
    ModuleIface::DataMap::iterator srcMapIt;

    // get pointer to object data vector
    if ( ( srcMapIt = dataMap.find( RDB_PKG_ID_OBJECT_STATE ) ) != dataMap.end() )
        objVec = srcMapIt->second;

    // do nothing if object data is not available
    if ( !objVec )
        return;

    RDB_OBJECT_STATE_t* ownObjectState = 0;

    for ( ModuleIface::DataVec::iterator srcVecIt = objVec->begin(); !ownObjectState && ( srcVecIt != objVec->end() ); srcVecI
    {
        RDB_OBJECT_STATE_t* objState = ( RDB_OBJECT_STATE_t* ) ( ( *srcVecIt )->data );

        if ( ( ( int ) objState->base.id ) == mOwnPlayerId )
        {
            ownObjectState = objState;
        }

    }
```

An alternative method for retrieving data is the routine `getEntryAtIndex()`. Let's assume you want to access the vehicle systems data of
your own vehicle from an input data interface. The following code fragment will be able to perform this job:

```
    RDB_VEHICLE_SYSTEMS_t* vehSys = 0;
    void* entry = 0;
    int i = 0;

    do
    {
        unsigned short flags = 0;
        entry = mDynamicsInData->getEntryAtIndex( RDB_PKG_ID_VEHICLE_SYSTEMS, i++, flags );

        if ( entry )
        {
            vehSys = ( RDB_VEHICLE_SYSTEMS_t* ) entry;
            if ( vehSys->playerId != ( unsigned int ) mOwnPlayerId )
                vehSys = 0;
        }
    } while ( entry && !vehSys );
```

Adding data to the output interface is also quite simple. The interface classes provide the corresponding
routine `addEntries(...)` which lets you add a set of data entries to the respective interface.

Example: add info about the sensor itself and about detected objects to sensor output iface

```
SensorPlugin::addSensorInfo()
{
    // first add info about the sensor itself
    RDB_SENSOR_STATE_t sensorState;
    memset( &sensorState, 0, sizeof( RDB_SENSOR_STATE_t ) );

    sensorState.id           = mId;
    sensorState.type         = ( mSensorType == video ) ? RDB_SENSOR_TYPE_VIDEO : RDB_SENSOR_TYPE_RADAR;
    sensorState.hostCategory = RDB_OBJECT_CATEGORY_PLAYER;
    sensorState.hostId       = mOwnPlayerId;
    :
    :
    mSensorOutData->addEntries( RDB_PKG_ID_SENSOR_STATE, ModulePlugin::RDBpkgId2size( RDB_PKG_ID_SENSOR_STATE, false ), false,

    for ( ModuleIface::DataVec::iterator srcVecIt = mOutputObjects->begin(); srcVecIt != mOutputObjects->end(); srcVecIt++ )
    {
        RDB_SENSOR_OBJECT_t sensorObject;
        memset( &sensorObject, 0, sizeof( RDB_SENSOR_OBJECT_t ) );

        RDB_OBJECT_STATE_t* objState = ( RDB_OBJECT_STATE_t* ) ( ( *srcVecIt )->data );

        if ( ( objState->base.id !=  ( unsigned int ) mOwnPlayerId ) ) // actually, own player should never be referenced
        {
            sensorObject.category    = objState->base.category;
            sensorObject.type        = objState->base.type;
            sensorObject.flags       = calcCriticality( mSensorOutData, objState->base.id, dist );
            sensorObject.id          = objState->base.id;
            sensorObject.sensorId    = mId;
            :
            :

            mSensorOutData->addEntries( RDB_PKG_ID_SENSOR_OBJECT, ModulePlugin::RDBpkgId2size( RDB_PKG_ID_SENSOR_OBJECT, false
        }
    }
    return true;
}
```

Workflow and dataflow in the plug-ins

The workflow of the plug-ins is quite simple:

- read data from RDB (handled by moduleManager)
- call the `update()` routine
  - parse incoming iface
  - add data to outgoing iface
- send resulting data via output port (handled by base class)

It's up to the user, what parts of the base classes will be overwritten, but usually it is enough to implement the plugin's `update()` routine.

Writing a custom sensor plug-in

As noted above, custom sensors must be derived from the class `Module::SensorPlugin`. At least the method `update()` must be implemented by the user. This routine is called by the ModuleManager in each simulation frame with the current frame number and a pointer to the interface class which contains all data that has been received from the TC.

Before calling the update()-method, the routines of the base class `SensorPlugin` will have performed the following steps:

- receiving of RDB data
- object filtering (by type)
- object detection (by frustum)

The following code fragment shows the main elements of the base class's update()-method and shall provide further insight into the methods / algorithms applied there.

```
int
SensorPlugin::update( const unsigned long & frameNo, Framework::Iface* data )
{
    // update the base class first
    if ( !ModulePlugin::update( frameNo, data ) )
        return 0;

    // allocate data structure for output data
    if ( !mSensorOutData )
    {
        std::string name = mName + std::string( "_OutData" );
        mSensorOutData   = new SensorIface( name );

        // set the reference in the base class
        setOutDataRef( mSensorOutData );
    }

    // has the simulation been reset?
    bool resetDetected = mSensorInData->mFrameNo < mFrameNo;

    mFrameNo = mSensorInData->mFrameNo;

    // now set the reference point from the given data
    if ( mNewFrame )
    {
        // set the basic output data
        setBasicOutData( mSensorInData );

        // get the player carrying the sensor
        RDB_OBJECT_STATE_t* ownObject = 0;
        bool extended = false;

        if ( mOwnPlayerId >= 0 )
            ownObject = mSensorInData->getPlayerObject( mOwnPlayerId, extended );
        else if ( !mOwnPlayerName.empty() )
            ownObject = mSensorInData->getPlayerObject( mOwnPlayerName, extended );

        // set the reference position of the sensor and calculate current position
        if ( ownObject )
        {
            mOwnPlayerId = ownObject->base.id;

            setReferencePos( ownObject->base.pos.x, ownObject->base.pos.y, ownObject->base.pos.z,
                             ownObject->base.pos.h, ownObject->base.pos.p, ownObject->base.pos.r );
            // calculate the current position of the sensor
            calcPos();

            // update the reference system from the given data
            updateRefSys();
        }

        // calculate the detection lists
        calcDetectionLists( ownObject );        // may be overwritten
    }

    // call the update routine of the derived class!
    int retCode = update( frameNo, mSensorInData );

    // work is done if we are not running a new frame
    if ( !mNewFrame )
        return retCode;

    // add further information in excess of the classic detection data for the RDB output channel
    prepareOutput( mSensorOutData, mSensorInData );         // may be overwritten

    // if RDB out interface is defined, then send the data
    sendRDB( mSensorOutData );

    return retCode;
}
```

The line `int retCode = update( frameNo, mSensorInData );` indicates the place where the update()-method of a custom implementation will be called. After this, the routine sendRDB() will send the detected objects via the RDB-out channel (2nd level RDB). File structure and contents correspond to the RDB that is established between TC and ModuleManager. By this, it is also possible to chain-link multiple instances of the ModuleManager.

By calling the method setFeedback() at the end of the update()-method, detected objects will be registered in the interface structure and will be sent from the ModuleManager to the TC. There, they will be used e.g. for the visualization of detection information (bounding frames). The feedback will only be sent if it has previously been activated (either in the configuration file or by using the method SensorPlugin::setFeedback()).

Another important method that may be overloaded by the user is `calcDectionLists()`. Here's again a code fragment of the base class that shall illustrate the principles of this method:

```
void
SensorPlugin::calcDetectionLists( RDB_OBJECT_STATE_t* ownObject )
{
    // filter objects by type
    filterObjects();

    // cull the objects and get resulting object list in interface
    cull();

    // calculate the occlusion of the objects
    calcOcclusion( mSensorOutData );

    // calc Object USK pos
    calcUSKPos( ownObject );

    // add object info relating to the sensor itself sensor
    addSensorInfo();
}
```

Writing a custom dynamics plug-in

Writing a custom dynamics plug-in is quite similar to writing a custom sensor plug-in. However, it should be considerably easier to do since complex routines like culling etc. are not
required. In the development environment, a sample plugin `SampleDyn` is provided for illustration.

Custom dynamics modules must be derived from the class `Module::DynamicsPlugin`. At least the method `update()` must be implemented by the user. This routine is called by the ModuleManager in each simulation frame with the current frame number and a pointer to the interface class which contains all data that has been received from the TC.

The following code fragment shows the main routines of the class `SampleDyn`:

```
int
SampleDyn::update( const unsigned long & frameNo , DynamicsIface* ifaceData )
{
    if ( !ifaceData || !mNewFrame )
        return 0;

    // check for the corresponding input structure
    RDB_DRIVER_CTRL_t* dynIn = ifaceData->getInputFromId( mOwnPlayerId );

    if ( !dynIn )
```

```
        fprintf( stderr, "SampleDyn::update: no input for player <%d> available\n", mOwnPlayerId );
    else
    {
        // get the inputs
        mInputSteering = dynIn->steeringTgt;
        mInputAccel    = dynIn->accelTgt;

        // compute the new position (see below)
        computeOutput();

        // check for availability of output data
        if ( !mDynamicsOutData )
        {
            fprintf( stderr, "SampleDyn::update: no output data available\n" );
            return 0;
        }

        // check whether player is already available
        bool extended = true;

        RDB_OBJECT_STATE_t* objState = mDynamicsOutData->getPlayerObject( mOwnPlayerId, extended );

        if ( !objState )
            objState = mDynamicsOutData->addPlayerObject( mOwnPlayerId, true );

        if ( !objState )
            fprintf( stderr, "SampleDyn::update: no output for player <%d> available\n", mOwnPlayerId );
        else
        {
            objState->base.id   = mOwnPlayerId;
            objState->base.type = RDB_OBJECT_TYPE_PLAYER_CAR;
            objState->base.geo.dimX = 4.60;
            objState->base.geo.dimY = 1.86;
            objState->base.geo.dimZ = 1.60;

            objState->base.geo.offX = 0.80;
            objState->base.geo.offY = 0.00;
            objState->base.geo.offZ = 0.30;

            objState->base.pos.x     = mInertialPos.getX();
            objState->base.pos.y     = mInertialPos.getY();
            objState->base.pos.z     = mInertialPos.getZ();
            :
            :

            // find the corresponding vehicle systems package in output interface
            RDB_VEHICLE_SYSTEMS_t* vehSys = 0;
            void* entry = 0;
            int i = 0;

            do
            {
                unsigned short flags = 0;
                entry = mDynamicsOutData->getEntryAtIndex( RDB_PKG_ID_VEHICLE_SYSTEMS, i++, flags );

                if ( entry )
                {
                    vehSys = ( RDB_VEHICLE_SYSTEMS_t* ) entry;
                    if ( vehSys->playerId != ( unsigned int ) mOwnPlayerId )
                        vehSys = 0;
                }
            } while ( entry && !vehSys );

            // not found? add to the output interface
            if ( !vehSys )
                vehSys = ( RDB_VEHICLE_SYSTEMS_t* ) mDynamicsOutData->addEntries( RDB_PKG_ID_VEHICLE_SYSTEMS, ModulePlugin::RDI

            if ( vehSys )
            {
                vehSys->playerId  = mOwnPlayerId;
                vehSys->lightMask = 0;
                vehSys->steering  = mSteeringAngle;

                if ( dynIn->flags & RDB_DRIVER_FLAG_INDICATOR_L )
                    vehSys->lightMask |= RDB_VEHICLE_LIGHT_INDICATOR_L;

                if ( dynIn->flags & RDB_DRIVER_FLAG_INDICATOR_R )
                    vehSys->lightMask |= RDB_VEHICLE_LIGHT_INDICATOR_R;
           }
        }
    }

    mLastFrameNo = mFrameNo;

    return 1;
```

```
}

void
SampleDyn::computeOutput()
{
    Framework::CoordFr deltaPos( mInertialSpeed.getX() * mDeltaSimTime + mInertialAccel.getX() * mDeltaSimTime * mDeltaSimTime
    deltaPos += Framework::CoordFr( 0.0, 0.0, 0.0, mDeltaSimTime * 0.1 );

    mInertialPos += deltaPos;

    mInertialSpeed.setX( mInertialSpeed.getX() + mInputAccel * mDeltaSimTime );
}
```

The RDB output data is sent automatically back to the TC using the `sendFeedback()` routine. This is done in the `update()` routine of the base class which is also the calling routine
for the derived class's `update()` routine.

Writing a custom driver plug-in

Writing a custom driver plug-in is similar to writing a custom dynamics plug-in. In the development environment, a sample plugin `DummyDriver` is provided for illustration.

Custom driver modules must be derived from the class `Module::DynamicsPlugin`. At least the method `update()` must be implemented by the user. This routine is called by the ModuleManager in
each simulation frame with the current frame number and a pointer to the interface class which contains all data that has been received from the TC.

The following code fragment shows the main routines of the class `DummyDriver`:

```
int
DummyDriver::update( const unsigned long & frameNo , DynamicsIface* ifaceData )
{
    if ( !ifaceData || ( mOwnPlayerId < 0 ) )
        return 0;

    if ( !mNewFrame )
        return 0;

    /* PART 1: collect and analyze incoming objects etc. --------------------------------------------------- */
    ModuleIface::DataMap::iterator srcMapIt;

    // get pointer to object data vector
    if ( ( srcMapIt = ifaceData->getDataMap().find( RDB_PKG_ID_OBJECT_STATE ) ) == ifaceData->getDataMap().end() )
        return 0;

    ModuleIface::DataVec* srcVec = srcMapIt->second;

    for ( ModuleIface::DataVec::iterator srcVecIt = srcVec->begin(); srcVecIt != srcVec->end(); srcVecIt++ )
    {
        ModuleIface::DataEntry *srcEntry = ( *srcVecIt );
        RDB_OBJECT_STATE_t* objState = ( RDB_OBJECT_STATE_t* ) ( srcEntry->data );

        fprintf( stderr, "DummyDriver::update: have object %s_%d at %.3lf / %.3lf / %.3lf\n",
                         objState->base.name, objState->base.id, objState->base.pos.x, objState->base.pos.y, objState->base.pos
    }

    /* PART 2: compose and send the feedback, i.e. the driver commands --------------------------------------------------- */
    RDB_MSG_t* msg = 0;

    RDB_DRIVER_CTRL_t* pInsData = ( RDB_DRIVER_CTRL_t* ) ModulePlugin::RDBaddPackage( ifaceData->mSimTime, ifaceData->mFrameNo

    if ( !pInsData || !msg )
        return 0;

    pInsData->playerId       = mOwnPlayerId;
    pInsData->steeringWheel  = 9.0 * sin( 0.8 * ifaceData->mSimTime );
    pInsData->throttlePedal  = 0.5 * ( 1 + sin( 0.8 * ifaceData->mSimTime ) );
    pInsData->brakePedal     = 0.0;
    pInsData->clutchPedal    = 0.0f;
    pInsData->gear           = RDB_GEAR_BOX_POS_D;
    pInsData->validityFlags  = RDB_DRIVER_INPUT_VALIDITY_STEERING_WHEEL | RDB_DRIVER_INPUT_VALIDITY_THROTTLE | RDB_DRIVER_INPU

    ModulePlugin::sendFeedback( msg, msg->hdr.headerSize + msg->hdr.dataSize );

    free( msg );

    mLastFrameNo = mFrameNo;

    return 1;
}
```

Sending and receiving SCP data

Module Plug-ins may also send and receive SCP data. For sending SCP messages, use the routine `bool ModulePlugin::sendSCPMessage( const std::string & text )`

The message will be sent immediately upon calling the routine.

Incoming SCP data will be forwarded to the method

`bool ModulePlugin::handleSCPCommand( const std::string & cmd )`

which may be overwritten by the user.

Working with OpenDRIVE data

Upon loading of a scenario, the corresponding OpenDRIVE file will also be propagated to the ModuleManager and, hence, will be available to the plug-ins. Reading the ODR file is
performed by the ModulePlugin class's routine

`bool readCourse( const std::string & filename, bool first = false )`

which may be overwritten (but doesn't have to).

If you want to access the OpenDRIVE data from within a plugin, just use the root node of the ODR data

`OpenDrive::RoadData* mRoadData`

which is available within the plugin.

The following example (excerpt of CrashSensor) shall illustrate the access to the OpenDRIVE data (here: static objects):

```
bool
CrashSensor::checkOfflineObjects( RDB_OBJECT_STATE_t* ownObjectState )
{
    // attach to existing OpenDRIVE data
    OpenDrive::OdrManager myManager;

    myManager.attach();

    // is the data really available?
    if ( !myManager.getRootNode() )
        return false;

    // go through all objects and check whether they are in the frustum
    OpenDrive::RoadHeader* roadHdr = reinterpret_cast< OpenDrive::RoadHeader* >( myManager.getRootNode()->getChild( OpenDrive:

    while ( roadHdr )
    {
        // are we in the bounding box of this roadHdr?
        if ( roadHdr->inBoundingBox( ownObjectState->base.pos.x, ownObjectState->base.pos.y ) )
        {
            OpenDrive::Object* odrObj = reinterpret_cast< OpenDrive::Object* >( roadHdr->getChild( OpenDrive::ODR_OPCODE_OBJEC

            while ( odrObj )
            {
                // check for crash
                :
                :
                if ( collision )
                    return true;

                odrObj = reinterpret_cast< OpenDrive::Object* >( odrObj->getRight() );
            }
        }
        roadHdr = reinterpret_cast< OpenDrive::RoadHeader* >( roadHdr->getRight() );
    }
    return false;
}
```

Access to another sensor's data

If you want to access another sensor's output data from within your own sensor (e.g. for sensor fusion), you may do so using the following code snippet (extracted from the perfect sensor's capability for object data fusion):

```
void
SensorPlugin::calcFusionData()
{
    static bool sVerbose = false;

    if ( !mSrcModuleVec.size() )
        return;

    // go through all source module's output data
    for ( std::vector< ModulePlugin* >::iterator it = mSrcModuleVec.begin(); it != mSrcModuleVec.end(); it++ )
    {
        SensorPlugin *srcPlugin = reinterpret_cast< SensorPlugin* >( *it );
        SensorIface* srcData = srcPlugin->getOutputData();
        if ( sVerbose )
            fprintf( stderr, "SensorPlugin::calcFusionData: checking data of sensor <%s>\n", srcPlugin->getName().c_str() );

        if ( srcData )
        {
            // fusion of object information only!
            ModuleIface::DataVec *srcVec = srcData->getDataVector( RDB_PKG_ID_OBJECT_STATE );

            if ( srcVec )
            {
                for ( ModuleIface::DataVec::iterator dvIt = srcVec->begin(); dvIt != srcVec->end(); dvIt++ )
                {
                    ModuleIface::DataEntry* srcEntry = ( *dvIt );

                    // does this entry already exist?
                    bool exists = false;

                    ModuleIface::DataVec *tgtVec = mSensorOutData->getDataVector( RDB_PKG_ID_OBJECT_STATE );

                    if ( tgtVec )
                    {
                        for ( ModuleIface::DataVec::iterator tgtIt = tgtVec->begin(); tgtIt != tgtVec->end() && !exists; tgtIt
                        {
                            ModuleIface::DataEntry* tgtEntry = ( *tgtIt );

                            RDB_OBJECT_STATE_t *srcObj = ( RDB_OBJECT_STATE_t* ) ( srcEntry->data );
                            RDB_OBJECT_STATE_t *tgtObj = ( RDB_OBJECT_STATE_t* ) ( tgtEntry->data );

                            exists = srcObj->base.id == tgtObj->base.id;

                            if ( !exists && ( tgtEntry->size == srcEntry->size ) )
                                exists = !memcmp( tgtEntry->data, srcEntry->data, srcEntry->size );
                        }
                    }

                    RDB_OBJECT_STATE_t *objData = ( RDB_OBJECT_STATE_t* ) ( srcEntry->data );

                    if ( !exists )
                    {
                        mSensorOutData->addEntries( srcEntry->type, srcEntry->size, srcEntry->flags, srcEntry->data );
                        if ( sVerbose )
                            fprintf( stderr, "SensorPlugin::calcFusionData: <%s> adding object <%s>\n", mName.c_str(), objData
                    }
                    else if ( sVerbose )
                        fprintf( stderr, "SensorPlugin::calcFusionData: <%s> rejecting object <%s>\n", mName.c_str(), objData-
                }
            }
        }
    }
}
```

For details of the sensor configuration in the moduleManager, see Sensor Fusion.

Compiling Plug-Ins on Ubuntu 16.04+

Following changes need to be done in the Qt project file in order to be able to compile plug-ins on Ubuntu 16.04+

```
QMAKE_LFLAGS += -fabi-version=3 -std=gnu++98
DEFINES += _GLIBCXX_USE_CXX11_ABI=0
```

This ensures that the compiler is creating code linkeable to the VTD Module Manager.

vtd317.png	(302 KB)	Marius Dupuis, 05.02.2016 19:22
sensorWedge01.png	(35.2 KB)	Marius Dupuis, 26.05.2016 08:16
myFrust.png	(697 Bytes)	Marius Dupuis, 26.05.2016 08:37
vtd372.png	(407 KB)	Marius Dupuis, 27.05.2016 12:15
vtd369.png	(272 KB)	Marius Dupuis, 27.05.2016 12:15
vtd351.png	(489 KB)	Marius Dupuis, 27.05.2016 12:16
vtd373.png	(240 KB)	Marius Dupuis, 05.06.2016 08:35
vtd374.png	(229 KB)	Marius Dupuis, 05.06.2016 09:22
vtd.2.0.3.addOns.MultiRaySensor.20170604.tgz	(20.3 KB)	Marius Dupuis, 04.06.2017 05:13