



# vsTASKER

## Tutorial

---

Step by Step Guide for New Users  
2021

## Table of Contents

# Table of Contents

vsTASKER Tutorial.....	6
Why vsTASKER.....	7
Compiler Setup .....	9
Hello World .....	10
Understanding the Basics.....	14
Architecture .....	15
C++ Programming .....	16
Browsing Samples .....	17
Console Samples.....	18
Attack.....	19
Drone Survey.....	22
Georeferenced Maps.....	25
Battleship .....	29
MainGun .....	32
OpenGL Samples .....	34
Radar Arrangement.....	35
Ballistic Missiles.....	37
ISS Orbiting .....	39
Indirect Fire.....	41
Air Traffic .....	42
Alternate Flight-Plan.....	45
OSG Sample.....	48
Bouncing Balls.....	49
Desert Flight .....	51
Helicopter Control.....	54
Suicide Attack.....	56
HLA Samples.....	58
Mäk Bounce.....	59
Pitch RTI Sample .....	64
HMI Sample.....	65
Sprites.....	66
Village .....	68
Integrated Samples.....	70
VegaPrime Sample.....	71
Pendleton.....	72
Google-Earth Sample .....	73
Takeoff .....	74
VBS-IG Sample .....	76
Anzac .....	77
Gamepad Settings .....	83
Infantry .....	86
VBS-3 Sample .....	90
Concept .....	91
IED .....	93
Corazol .....	97
Titan Sample .....	102
Waiouru .....	103
Qt Sample.....	106
Train Doors.....	107
Matlab Sample.....	110
Data Plotting.....	111

## Table of Contents

Simulink Sample .....	112
STK Samples.....	113
Communications .....	114
Data Provider .....	117
Active/X STK Viewer.....	119
Delta 3D Sample .....	120
Little Town .....	121
Di-Guy Sample .....	122
First Shooter.....	123
GL-Studio Sample .....	125
Altimeter.....	126
RotorLib Sample.....	127
VR-FORCES Sample .....	128
STAGE Sample .....	130
Flsim Sample.....	132
Helisim Sample.....	134
Trinigy Sample.....	136
My First Simulations .....	137
Simple Entity.....	138
Changing Parameters.....	146
Cloning an Entity.....	148
Using Plans .....	150
Plan at Design.....	151
Runtime Plans .....	154
Simple Logic .....	157
Simple Component .....	162
Sub-banding.....	171
Simple Knowledge .....	172
Simple HMI .....	177
Input/Output Sprites.....	178
Sprite Model .....	184
Using OpenSceneGraph.....	190
Going Further.....	191
DIS .....	192
HLA .....	193
Concepts.....	194
Simple Example .....	197
Setting the RTI.....	198
Mäk RTI .....	200
Pitch RTI .....	201
Defining SOM.....	202
Using Direct Access .....	205
Publish .....	206
Subscribe .....	213
Using DataModel .....	216
Publish .....	226
Subscribe .....	229
Running the Sample .....	231
My Own Federation.....	233
Defining the FOM.....	234
Defining the FedItems.....	238
Publish.....	239
Subscribe .....	246
Running my Federation .....	251

## Table of Contents

Complex Federations .....	253
RPR-FOM .....	254
Importing SOM .....	278
CIGI .....	279
Concepts .....	280
Definition Files .....	282
Property Window .....	284
First Steps .....	288
Messages .....	297
Host to IG .....	300
IG to Host .....	303
Testing with MPV .....	305
Using VBS-IG .....	309
Using Unigine .....	312
LAN .....	315
Integration with third-parties .....	317
VBS-IG .....	318
Scenario Setup .....	319
Map Extraction .....	325
Elevation Layer .....	335
Adding Entities .....	340
Setting up the Tank .....	348
Detect and Fire .....	350
VBS-3 .....	354
Concept .....	355
Adding Entities .....	357
Simple Logic .....	360
Titan Vanguard .....	363
Concept .....	364
Scenario Setup .....	366
Adding Entities .....	368
Firing a Weapon .....	371
VegaPrime .....	373
CIGI Simple .....	377
STK .....	378
Simple .....	381
Create STK Scenario .....	382
Importing Scenario .....	383
Running the Simulation .....	386
Updating the Scenario .....	387
Detections .....	390
MATLAB .....	395
Simulink .....	397
GL-Studio .....	400
SQL .....	404
Using mySQL .....	405
mySQL Sample .....	411
Simple Test .....	412
Using MariaDB .....	414
Using SQLite .....	418
Qt .....	419
Simple Remote .....	426
Qt + HMI .....	430
Qt + OSG .....	431

## **Table of Contents**

Copyright .....	433
-----------------	-----

# **vsTASKER Tutorial**

This tutorial is intended to software/simulation engineers or decision makers who need to use a simulation development kit to develop a platform to test their models, immerse some real equipments into a synthetic environment, build a flight simulator or any other trainer for complex devices or simply need a real time workbench to integrate various software devices.

It is divided into two parts.

The [Samples](#) chapter list various demos trying to cover several aspects and domain of the product. If not all of them have to be tried, they give essential hints on what the product is about and how things can be done, using process flow charts (logics) and runtime models (components).

The [Do it Yourself](#) chapter is intend to let you build your own simulation by creating scenarios, adding entities, designing logics, writing components and assembling all these to produce a runtime solution.



*The Integration chapter is for information only. The demo distribution does not contain the associated databases. They are available in the evaluation and release versions only.*

This manual does not replace the training course (basic or advance) and is mainly intended to provide a guidance (or a reminder) for a good understanding of the product.

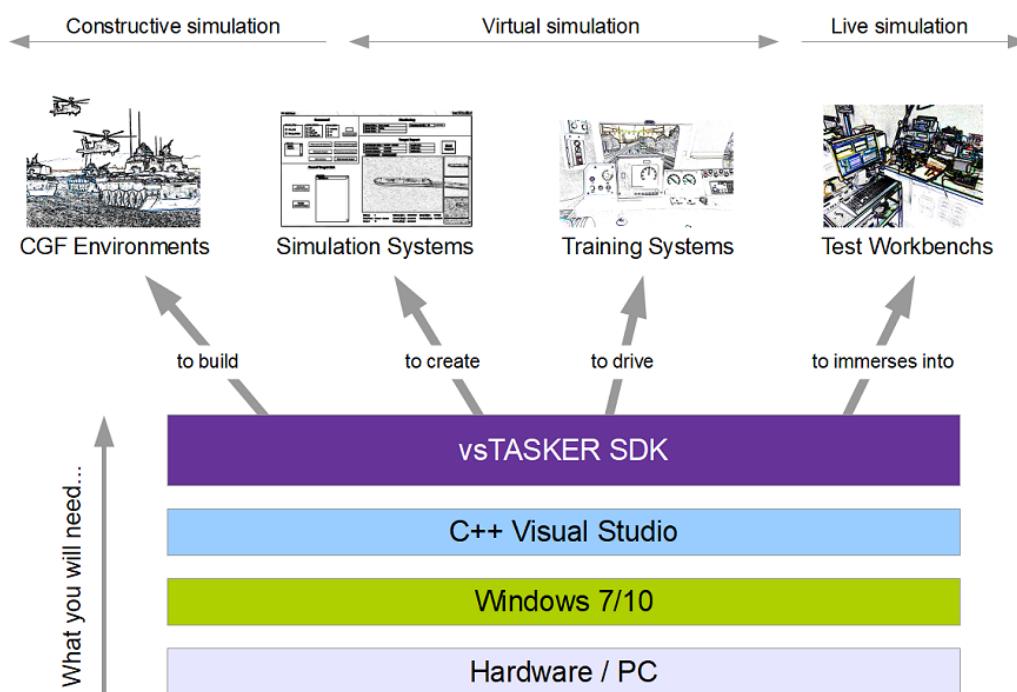
# Why vsTASKER

- **Where it is good at**

vsTASKER allows study, animation, and visualization of any size and complexity of scenarios. Using flowcharts and graphical paradigm and automatic C++ code generation, user can build a simulator with unmatched simplicity. vsTASKER does not limit to any particular field (see some projects)

Targeting software engineers and system integrators, it provides under one roof everything needed to develop a simulation infrastructure, maintainable and scalable. It covers a wide range of needs, from Defense, mission planning, CGF, electronic warfare, UAV, train cabin simulators, railways and ATC trainers (...)

After 13 years on the market worldwide, vsTASKER has proven its value under high technical constraints and tight schedules (see our testimonials).



vsTASKER provides unprecedented mechanisms to tailor your synthetic environment, either from a simple 2D plan map to the most complex 3D game like environment. Once you have understood the concepts, testing a scenario or developing a system under vsTASKER becomes so straightforward you will wonder how you did before. vsTASKER has been designed after years of consultation with

## **Why vsTASKER**

customers struggling with either too low-level compilers or specialized high-end overkill products.

Instead of reinventing the wheel by writing a framework from scratch or trying to break into pieces a complex expensive software to fit it into your needs, vsTASKER provides all the necessary mechanisms for real-time simulation without overwhelming you with tons of specific capabilities and parameters.

## **• Unleash the power of a framework**

While a end-of-line product only gives you what it has, a framework helps you to get what you want. See how vsTASKER main features can help you doing things by yourself:

- Logic defined visually using Grafcets and State-Charts;
- Knowledge based on Facts and user Rules;
- Behavior-Trees to trigger Logic and Knowledge;
- Component/Device models to provide capabilities;
- Routines as piece of code dropped on the map;
- Visual feedback at runtime;
- Runtime mission planning;
- Predefined material ready to be used;
- OpenScenGraph & osgEarth ready;

# Compiler Setup

vsTASKER needs a compiler to produce the simulation engine.

Visual Studio compiler is the default supported one.

Libraries from vc90 to vc110 are provided. According to the Visual Studio version chosen, vsTASKER will link against the proper libraries.

- **How to get Visual Studio**

You can use Visual Express 2008 or 2010 to start. This environment is much faster and has a lower memory footprint than the newer versions of Studio.

Nevertheless, with Express, you will need to install the SDK on top of it if not provided.

- **Demo Version**

The demo/free version provides Visual Express 2010. Installation is proposed during full install.

You can manually install it from the Installation window panel or, once product installed, opening one of the following files:

- Tools/VCExpress/setup\_web.exe
- Tools/VCExpress/autorun.exe

- **Licensed Version**

vsTASKER supports several installations of Visual Studio:

- **v6: 2009 (vc90), 2010 (vc100), 2012 (vc110), 2017 (vc141, x64)**
- **v7.0: 2010 (vc100, x32, x64), 2015 (vc140, x32, x64), 2017 (vc141, x64)**
- **v7.1: 2010 (vc100, x32, x64), 2017 (vc14, x32, x64)**

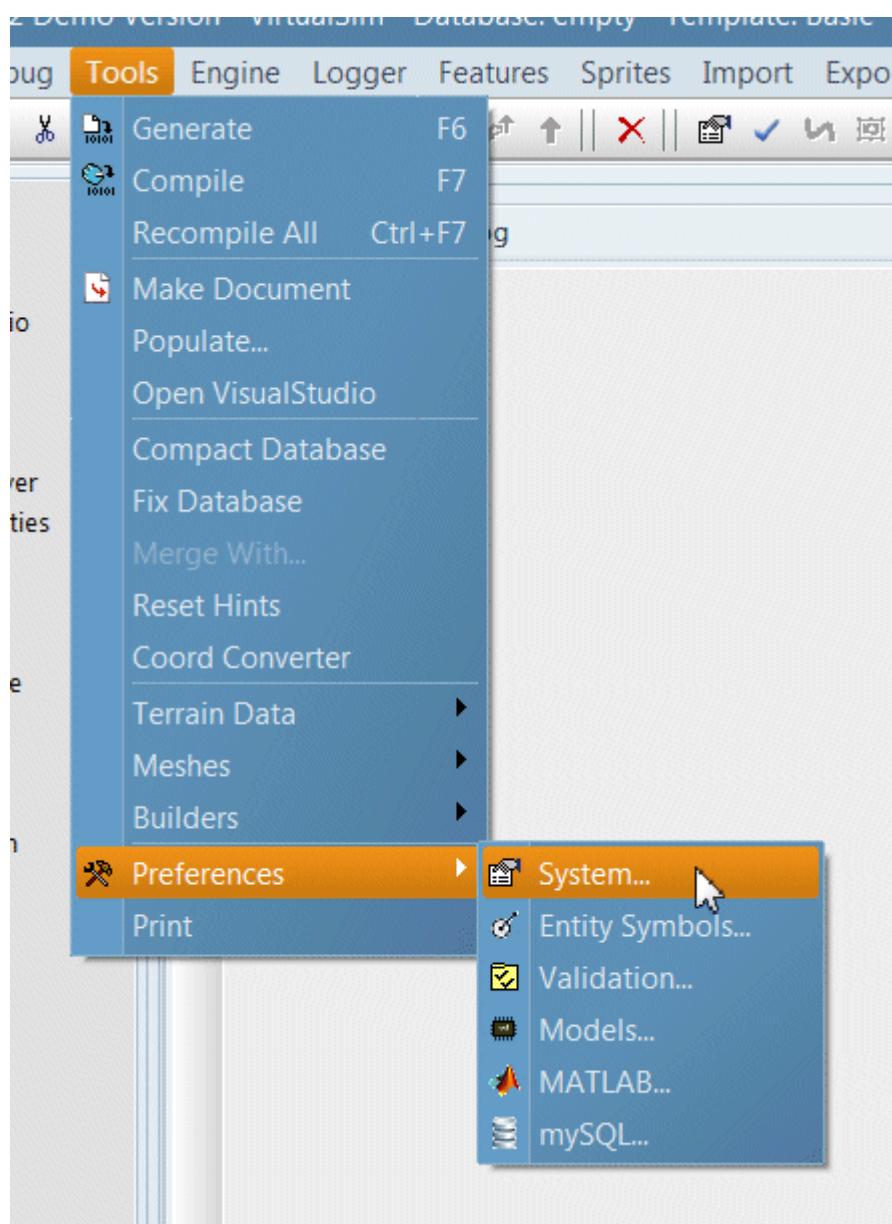
## Hello World

# Hello World

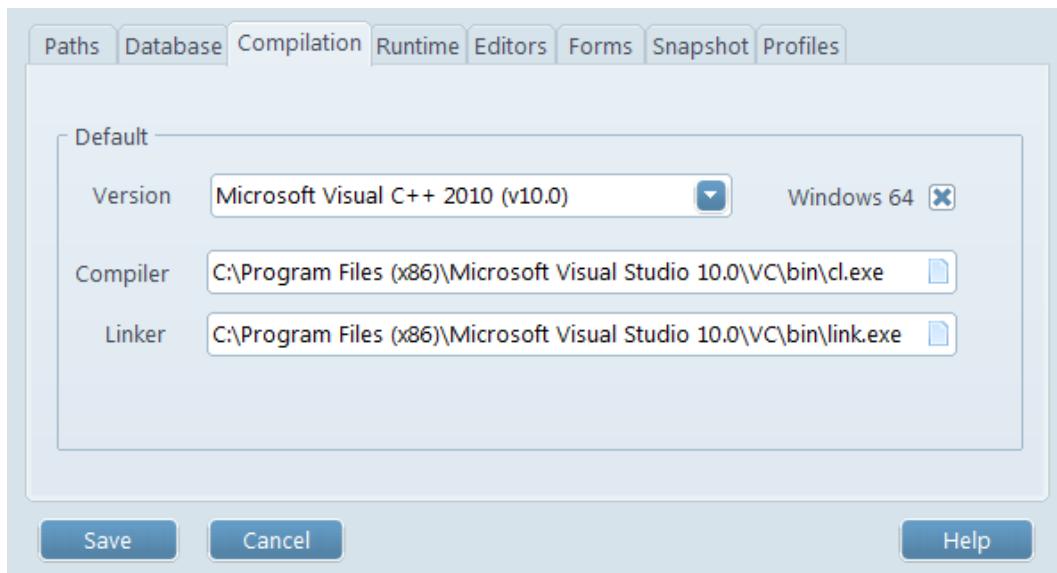
This first exercise will also test the configuration.

First, **check the compiler**.

Open the preferences window:



Then check that Visual Studio 2010 (vc100) is set (for the demo version, as Express 2010 is provided in the install). Otherwise, make sure that you select here the compiler you want to use.

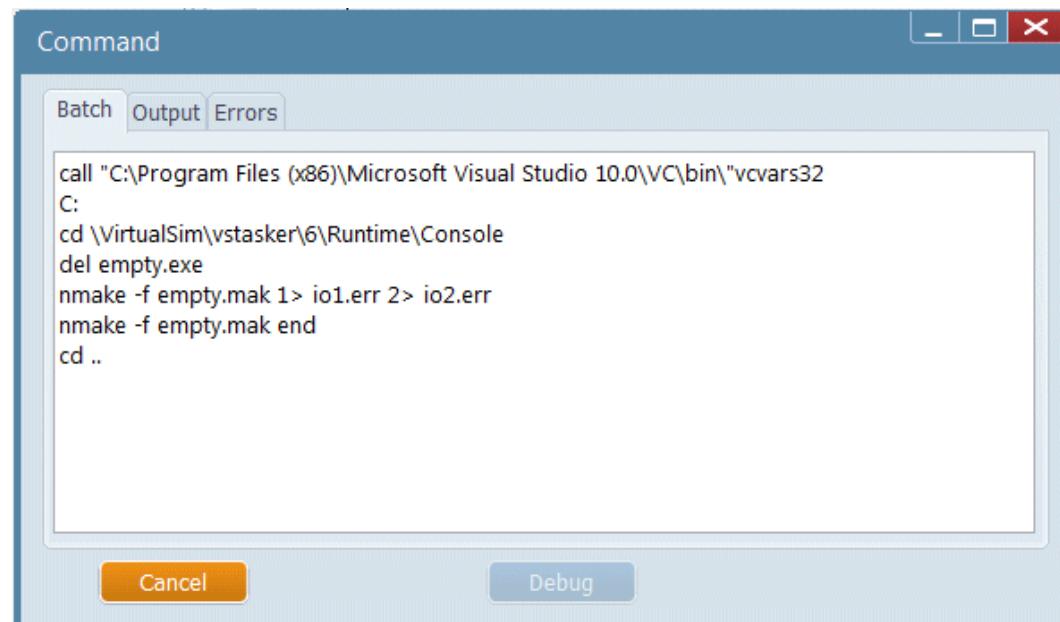


If Visual Studio has been installed on another partition, make sure to change it on the Compiler and Linker text fields. Save.

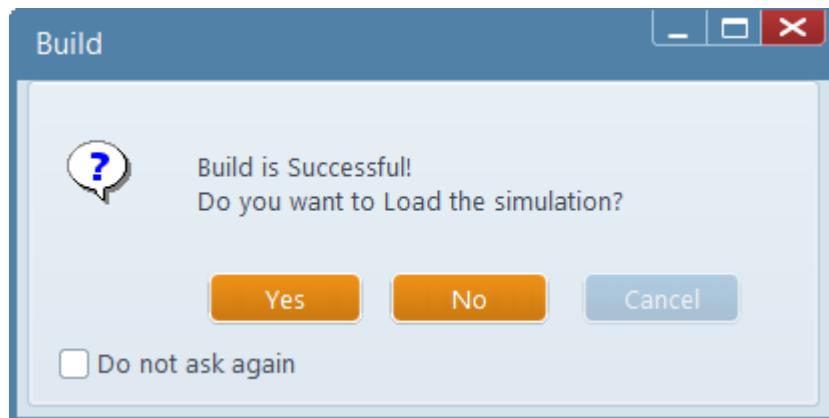
Then,

Open the database [empty.db](#) in Data/empty directory.

Recompile:



## Hello World



Press YES.

You should then get this console window:

```
C:\Windows\system32\cmd.exe
C:\VirtualSim\vsTasker\6>C:
C:\VirtualSim\vsTasker\6>cd \VirtualSim\vsTasker\6\Runtime\Console
C:\VirtualSim\vsTasker\6\Runtime\Console>start /b empty.exe
=====
=           vsTASKER 6.2.2          =
=                                     =
= Copyright (c) VirtualSim Sarl. 2004-2018 =
= All rights reserved. International Patent. =
= Simulation Engine is license free! =
=                                     =
=     www.virtualsim.com - Nice - France =
=           support@virtualsim.com =
=====

Having loaded environnement for: empty
Simulation Engine is ready
-
```

Back to vsTASKER GUI, start the simulation:



In the Console window, you should see the following:

```
Having loaded environnement for: empty
Simulation Engine is ready
player init
player reset
Simulation started!
1 [Base: 30 Hz - Requested: 30 Hz - Effective: 32 Hz ($Sleep=31)]
2 [Base: 30 Hz - Requested: 30 Hz - Effective: 31 Hz ($Sleep=31)]
3 [Base: 30 Hz - Requested: 30 Hz - Effective: 31 Hz ($Sleep=31)]
```

meaning everything is fine and you are ready to proceed.

If you cannot go through this simple test, please to send us a message with a description of your problem and the support team will fix it quickly:

[support@virtualsim.com](mailto:support@virtualsim.com)

## **Understanding the Basics**

# **Understanding the Basics**

TODO>: Insert description text here... And don't forget to add keyword for this topic

# **Architecture**

<TODO>: Insert description text here... And don't forget to add keyword for this topic

## C++ Programming

# C++ Programming

<TODO>: Insert description text here... And don't forget to add keyword for this topic

## Browsing Samples

vsTASKER provides ready to run scenarios (already compiled) that try to cover most of the features of the product.

It is a good idea to try most of them to learn how things are done inside a scenario to perform specific actions.

If you have installed Visual Express 10 included into this distribution, you can recompile all the samples.

This is not mandatory. To load the simulation engine associated with each database, click the  button on the toolbar or select menu [Engine::Launch](#).



*If the  button seems ineffective, use the [Engine::Detach](#) call then try again. If still fails, recompile.*

Then, go to [My First Simulations](#) to try building your own.

Let's see some of them now.



*The Demo version has less samples than the normal release of the product.  
They are all located in <C:/VirtualSim/vsTasker/6/Data/Db/Samples>*

## **Console Samples**

# **Console Samples**

Console simulations can run on any computer and can also be easily and freely distributed.  
Most (if not all) of them are using vsTASKER GUI for output during runtime.

They are the simplest and also the fastest simulation code vsTASKER can generate.

You can open some samples in [\*\*data/db/samples/console\*\*](#)

# Attack

Let's open **attack** sample.

Select  in the main toolbar, then open **attack.db** in **data/db/samples/console/attack**

- **Description**

This example shows 2 things:

1- How to create 100 entities randomly from a catalog.

See the Logic **createEntity** for that. This logic is attached to the scenario player.

2- A simple logic that triggers the following behavior:

- detects foes
- select one random if not already attacking or being attacked
- show line of attack
- invite target to attack (duel)
- fight and randomly kill
- reengage after fight

Here, three logics have been defined.

One is given to the scenario Player. It is intended to create all the entities at start, randomly on the gaming area:

```
WCoord pos(RANDOM(-200,200),RANDOM(-200,200),0);
Entity* basic = new Entity("basic", pos);

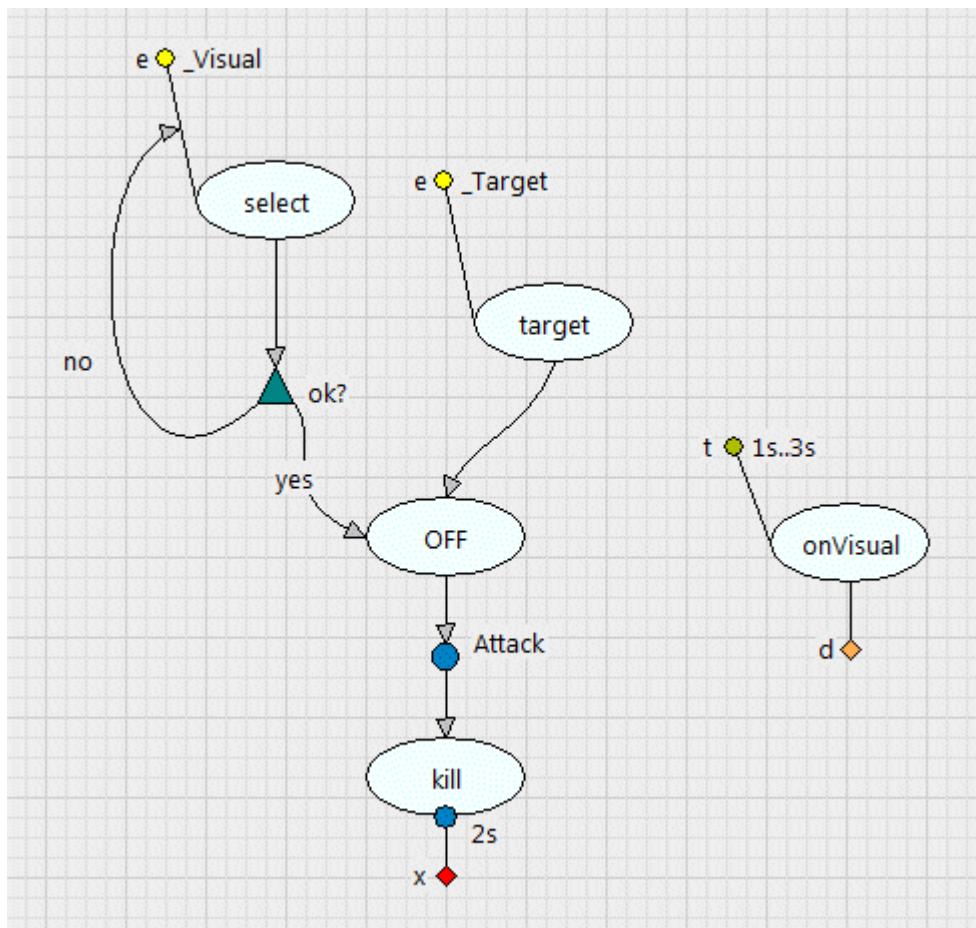
nb++;
printf("%d\n", nb);

if (nb >= L:numb_ent) return DONE;

return AGAIN;
```

The second logic **Attack** is given to each entity on the game play.

## Attack



The principle is simple:

Whenever an entity detects something around it (`_Visual` event), it selects randomly one "target" which is not already in a duel, then informs this target about a duel by sending it an event `_Target`.

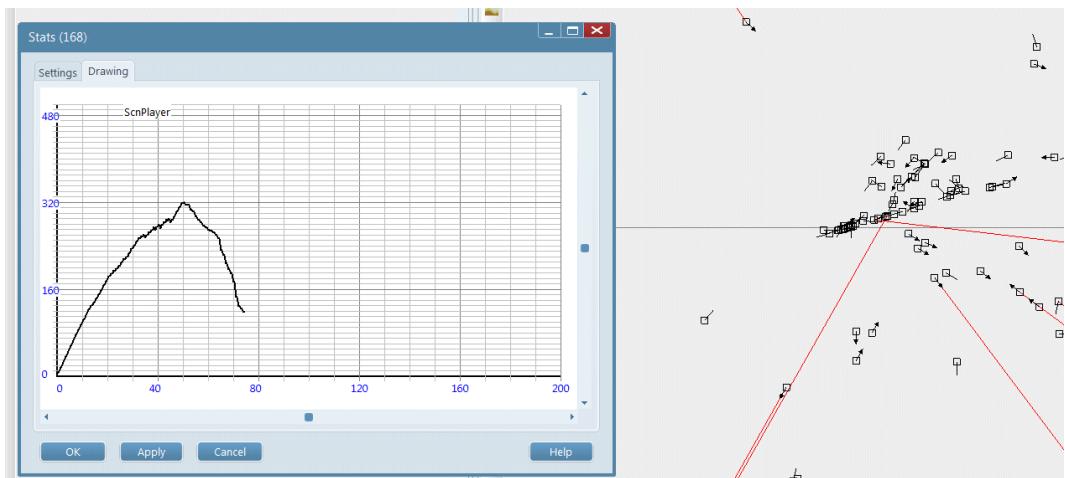
This events switch off the visual sensor then start the Attack. One of the two entities will be killed. The winner will restart the logic and fight again.

The scenario stops when only one entity is left.

## • Runtime

During the scenario, a graph shows the number of entities fighting in duels.

## Attack



## Drone Survey

# Drone Survey

Let's open **drone** sample.

Select  in the main toolbar, then open **drone.db** in **data/db/samples/console/drone**

## • Description

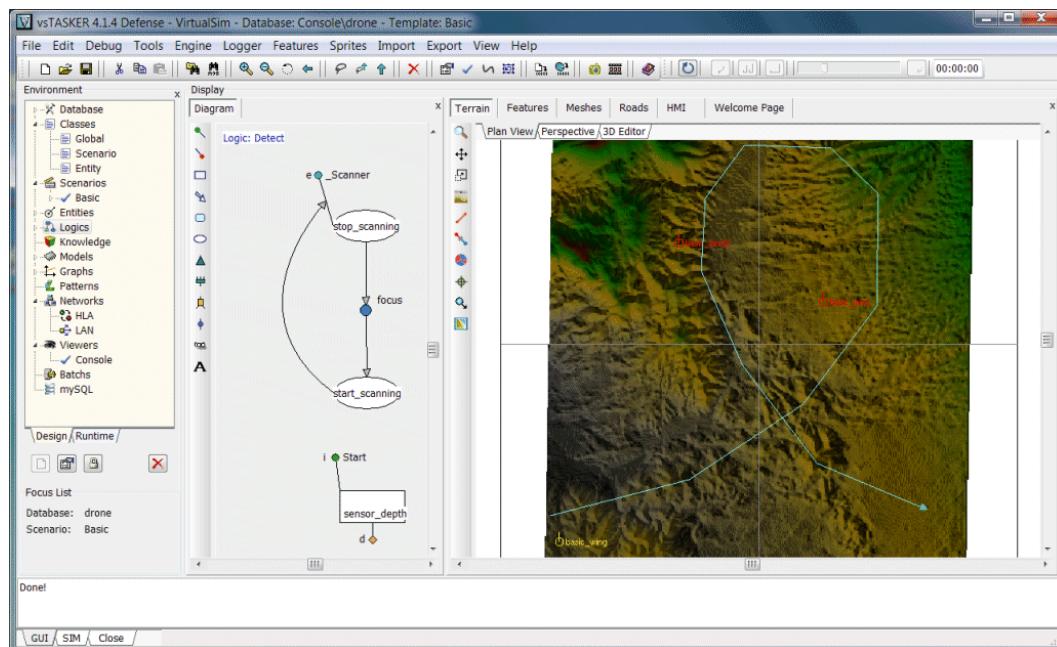
In this example, you have a drone following a trajectory and scanning the ground below it with a short view sensor which swipe the surface.

The logic is described in Detect. As soon as the scanner detects something, it stop scanning and focus on the thing detected, until a certain distance is reached (4000 m), then it starts scanning again.

The 4000 value is hard coded into the focus delay object of the logic. This can easily be set elsewhere so that the user can change it without recompilation.

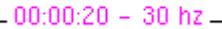
## • Runtime

Once loaded, your application will looks like below:

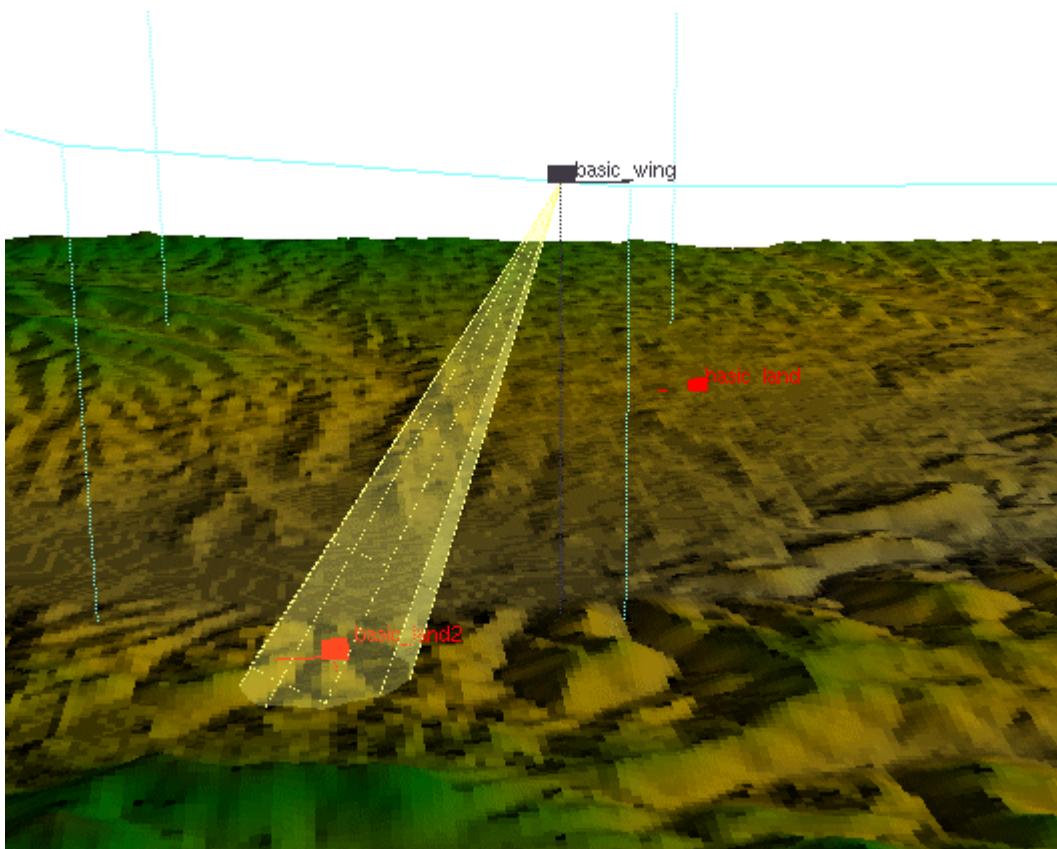


Load the simulation engine with the button  . A console application should pop.

Start the simulation with 

You can tell that the simulation is running because of  information displaying the current simulation time and the frequency. This frequency is the simulation engine one (in the console) and not the GUI that runs at a much lower rate.

Now, select the perspective view using 



You should be able to move your camera view point with the mouse, holding the left button.

Zooming in and out can be achieved with the roll know.

To increase the simulation time, just slide the cursor:  to the right. Check on the map display the frequency the system can reach. The more CPU power, the fastest the simulation can go without losing accuracy.

## **Drone Survey**

Stop the simulation using 

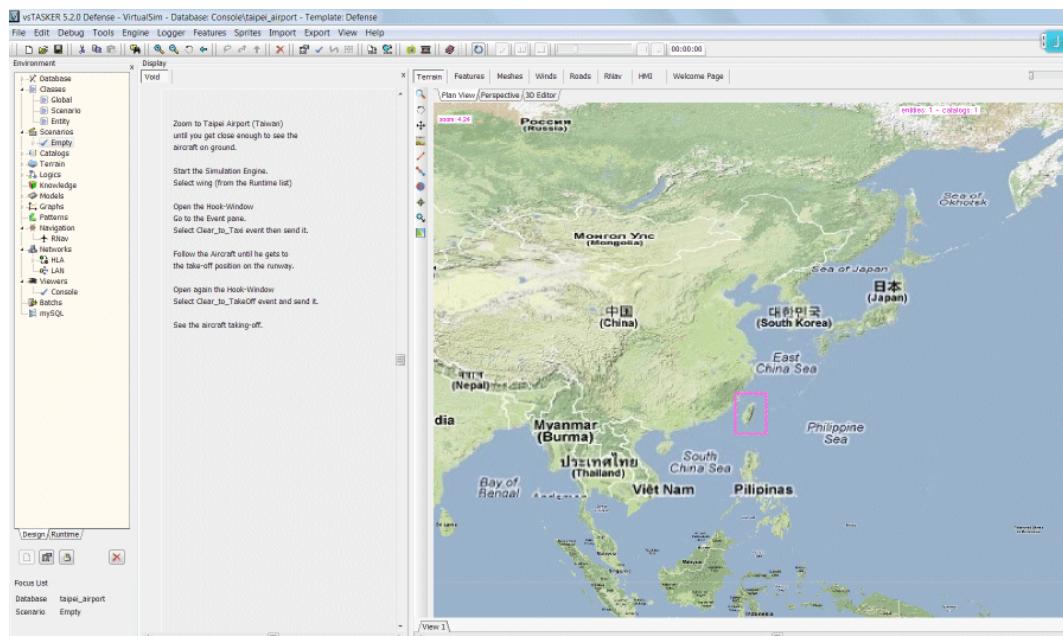
# Georeferenced Maps

Let's open [taipei\\_airport](#) sample.

Select  in the main toolbar, then open [taipei\\_airport.db](#) in [data/db/samples/console/taipei\\_airport](#)

vsTASKER supports multiple level of detail zooming on tiled map that are georeferenced, in order to provide accurate simulation on very big areas or by combining high level accuracy on certain areas while maintaining a low resolution on others.

Once loaded, your application will looks like below:

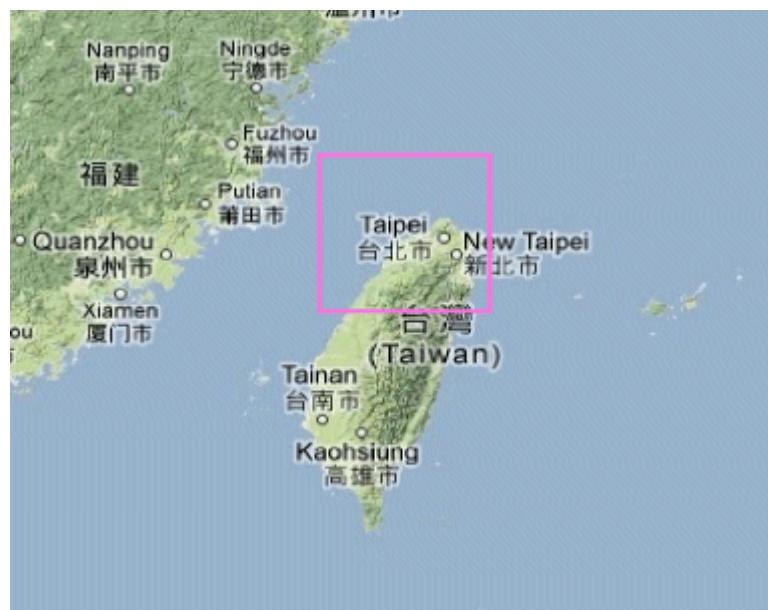


Load the simulation engine with the button  . A console application should pop.

Start the simulation with .

As nothing really happen, we must zoom strongly on the Taiwan island and precisely, on Taipei Airport.

## Georeferenced Maps



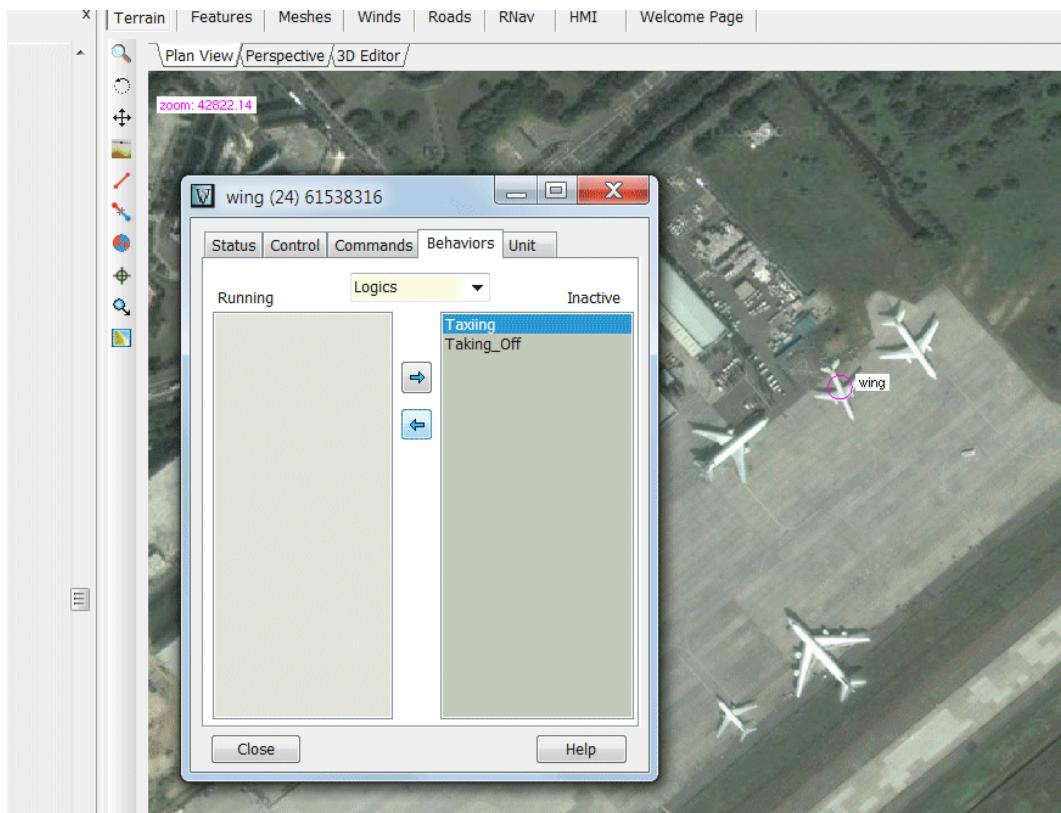
## Georeferenced Maps



You will see that the zooming in will automatically increase the resolution, as the raster database already got the raster loaded for this specific area. vsTASKER is not streaming the web nor accessing a huge worldwide database. This area is already available offline into [Data/Map/Terrains/taipei\\_airport.map](#)

Now, double-click the aircraft between the two other to popup the hook-window:

## Georeferenced Maps



Then in the Hook-Window, select the **Behaviors** pane, then select **Taxiing** Logic and use the button to move the Logic from *Inactive* to *Running*. The **wing** Entity will start **taxiing**.

You can move the map using the Ctrl key depressed while moving the mouse with the left button down.  
Once the Entity aligned on the runway, select **Taking\_Off** Logic and activate it the same way, for **take-off**.

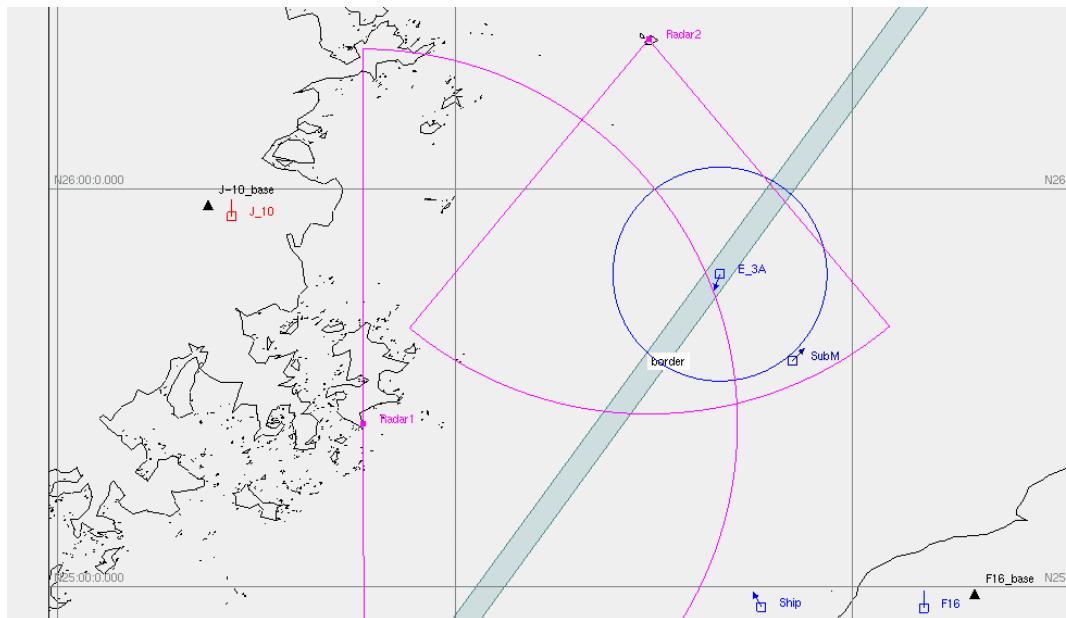


# Battleship

Let's open **elecwfare** sample.

Select  in the main toolbar, then open `taipei_airport.db` in `data/db/samples/console/elecwfare`

This demo uses some complex missions and covers many aspects of vsTASKER.



Radar1 & Radar2 are scanning the sea between China and Taiwan.

E3-A entity is located on the border (gray area).

When the E3-A is detected by both Radar1 & Radar2, (see J10\_logic, radar\_detection), J10 will take-off and go for interception.

When J10 will be detected by E3-A, information will be sent to Ship that will try to protect E3-A by sending a missile against the intruder detected by E3-A.

At the same time, F16 will take-off toward E3-A for close protection.

When close enough to target, J10 will fire missile against E3-A and return immediately to Base as soon as E3-A is destroyed.

F16 will normally chase for air combat but will not cross the border, as specified in the F16\_logic

## Battleship

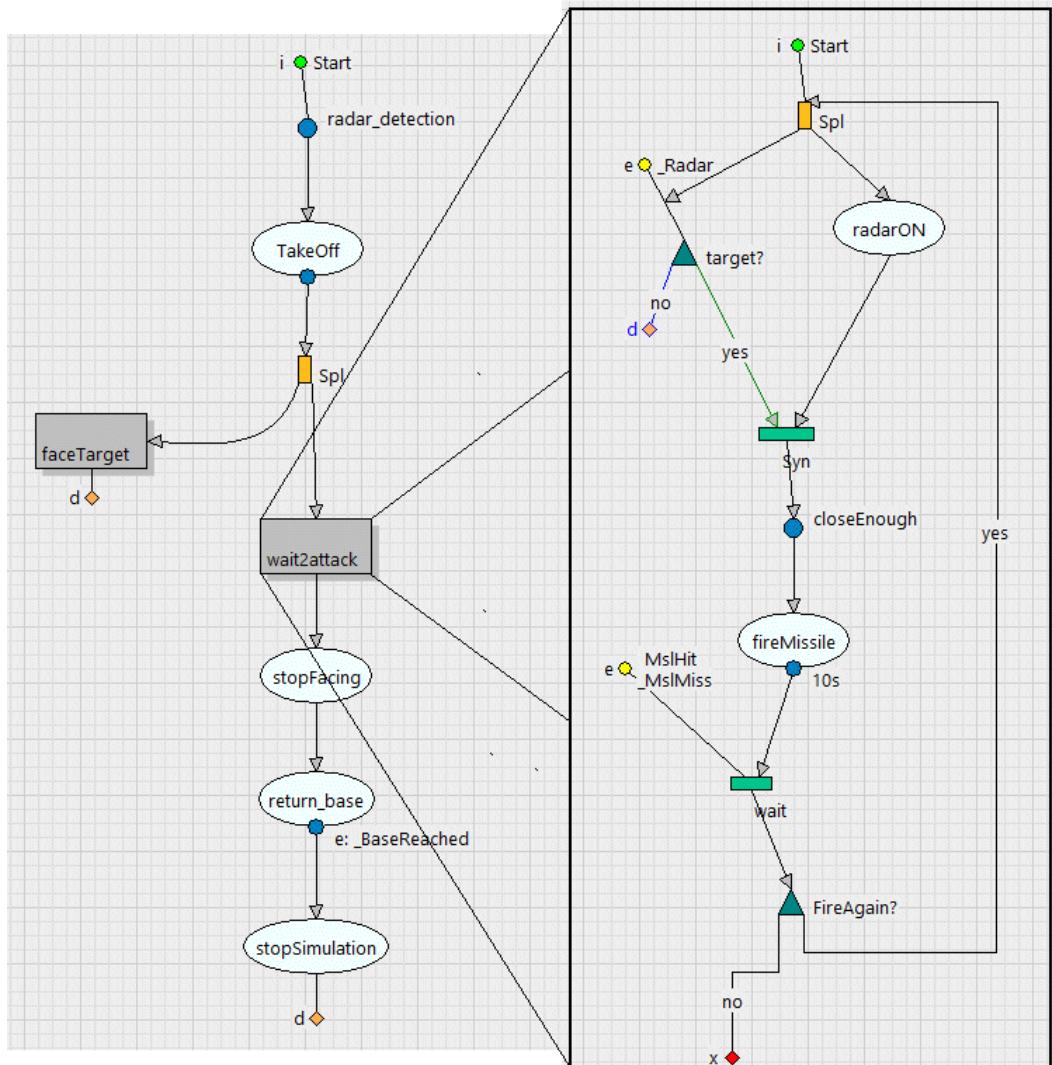
You can run this demo several times but you better accelerate the time to x16

You can also try the Batch mode to see how fast vsTASKER can run on such a simple simulation case study (turn speed to maximum)

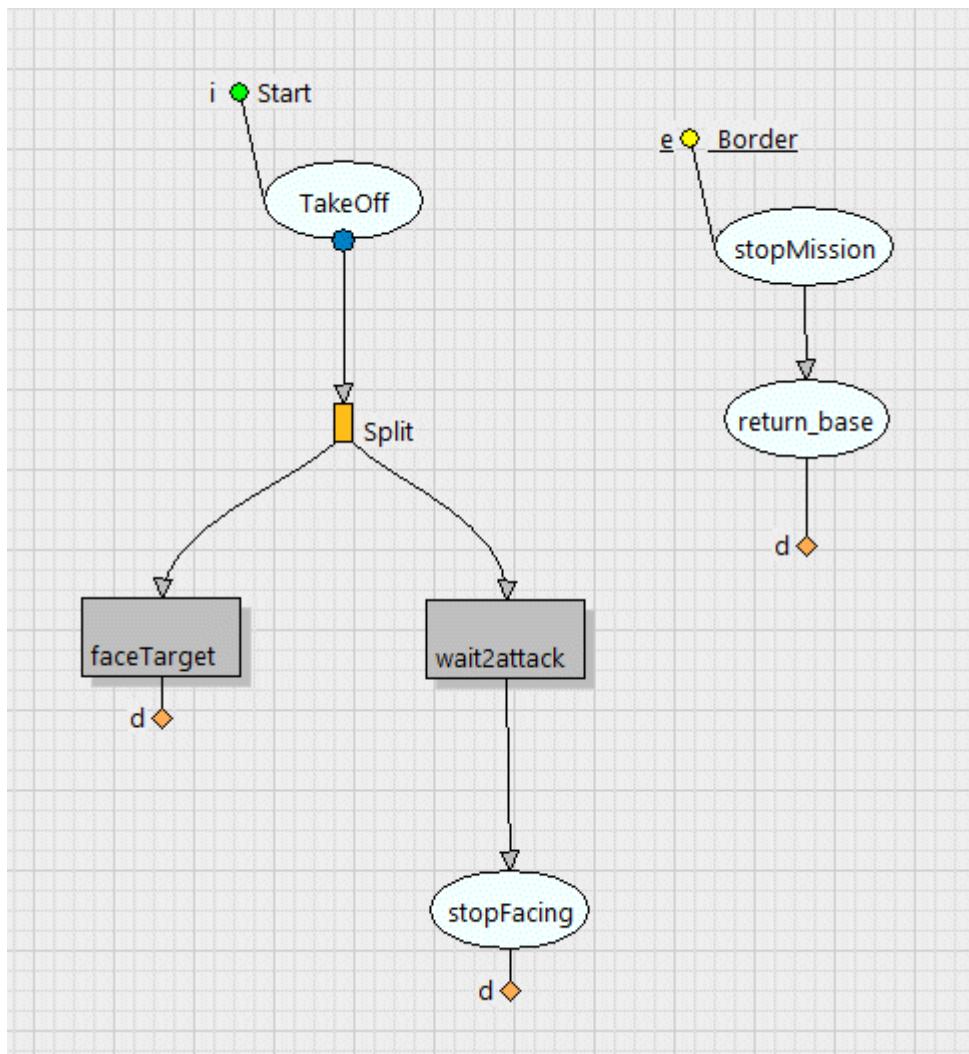
## • Logics

They are two logics which are interesting in this demo:

1. The J\_10 one, which controls the logic of the fighter from radar detection to missile fire and back to the base
2. The F16 one which controls the logic of the defender interceptor



J\_10 Logic



F\_16 Logic

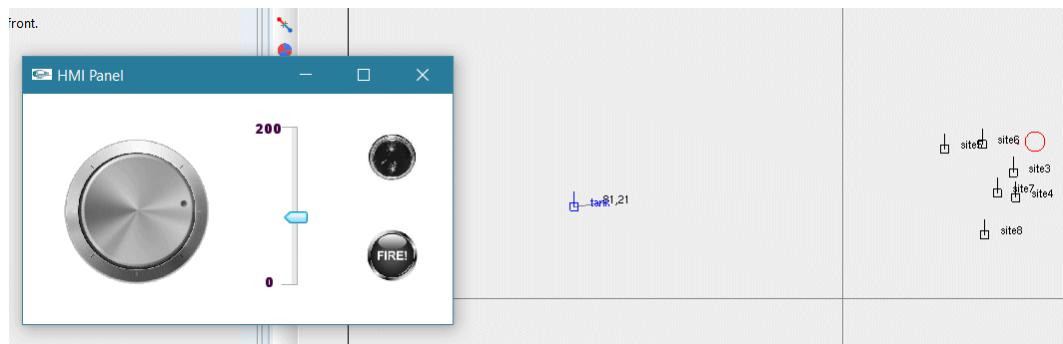
## MainGun

# MainGun

Let's open **main\_gun** sample.

Select  in the main toolbar, then open `taipei_airport.db` in `data/db/samples/console/main_gun`

This sample shows how to use Commands or HMI to control a tank main-gun weapon against a target.



## • How to Use

Start the simulation.

Click on the HMI window to make it appear in front.

Use the rotator to orient the main gun.

Use the vertical slider to elevate the main gun

Fire using the OFF light

Curve window will appear.



## **OpenGL Samples**

# **OpenGL Samples**

OpenGL simulations can run on most computer with a good and can also be easily and freely distributed without the GUI, as a standalone application.

OpenGL runtime application replicate the GUI mapping with some restriction.

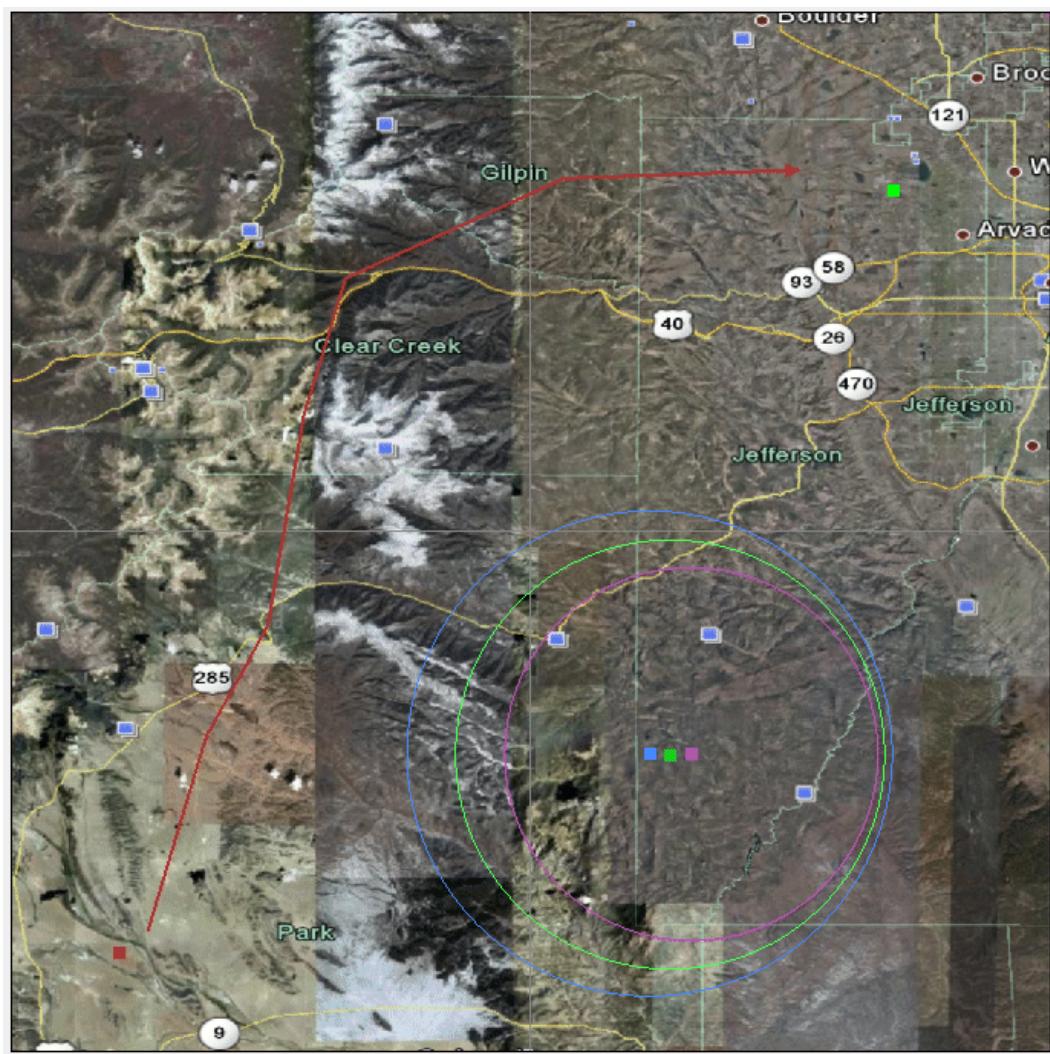
You can open some samples in **data/db/samples/opengl**

# Radar Arrangement

Let's open [radar\\_arrangement](#) sample.

Select  in the main toolbar, then open `radar_arrangementdb` in `data/db/samples/analysis/radar_arrangement`

Once loaded, your application will looks like below:



Monte Carlo simulation trying to position the three radar to detect at best the incoming missile doing terrain hugging flight.

Missile speed is 300m/s.

Radar detections are subject to terrain occultation (LOS).

The batch object computes the position of the radars at each iteration according to some predictive rules.

Compile and run the simulation.

## Radar Arrangement

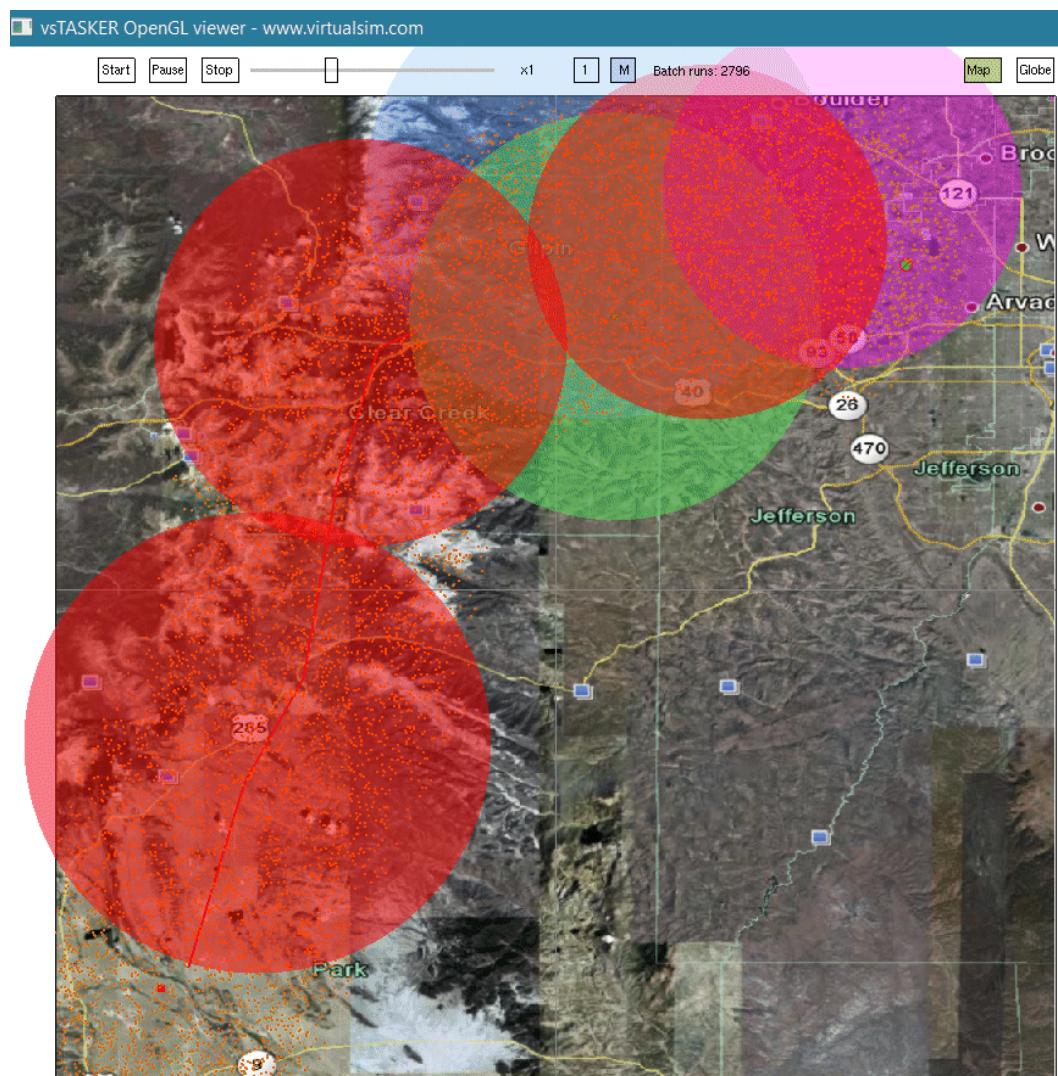
When the OpenGL windows is displayed, start the simulation.

At each run, vsTASKER will position the three radars according to various criteria then, the missile will be launched and it will follow the trajectory on the horizontal axis, and the terrain on the vertical axis. Detection time will be added to rank the radar positioning.

Click the M button (top toolbar) to run the simulation at maximum speed.

During the simulation, the best position are shown it red.

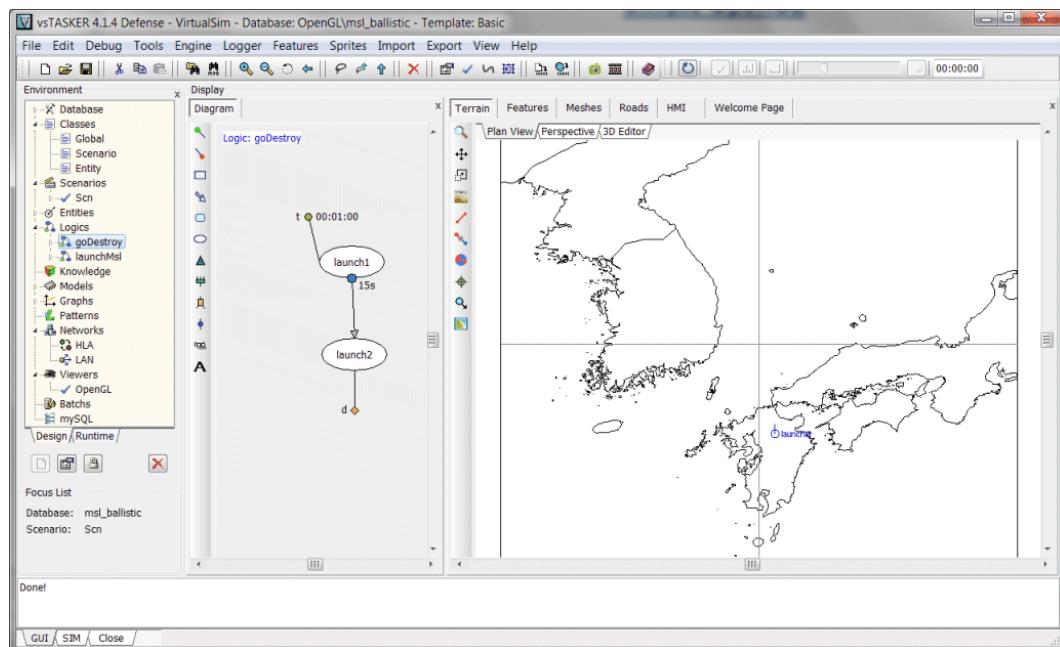
All the dots show the radar position tested.



# Ballistic Missiles

Let's open `msl_ballistic` sample.

Select  in the main toolbar, then open `msl_ballistic.db` in `data/db/samples/opengl/msl_ballistic`. Once loaded, your application will look like below:



Load the simulation engine with the button . A white OpenGL window should pop.

Start the simulation with , then select the "vsTASKER OpenGL viewer" application. This simulation is quite slow because area is huge. You can increase the simulation speed. To do so,

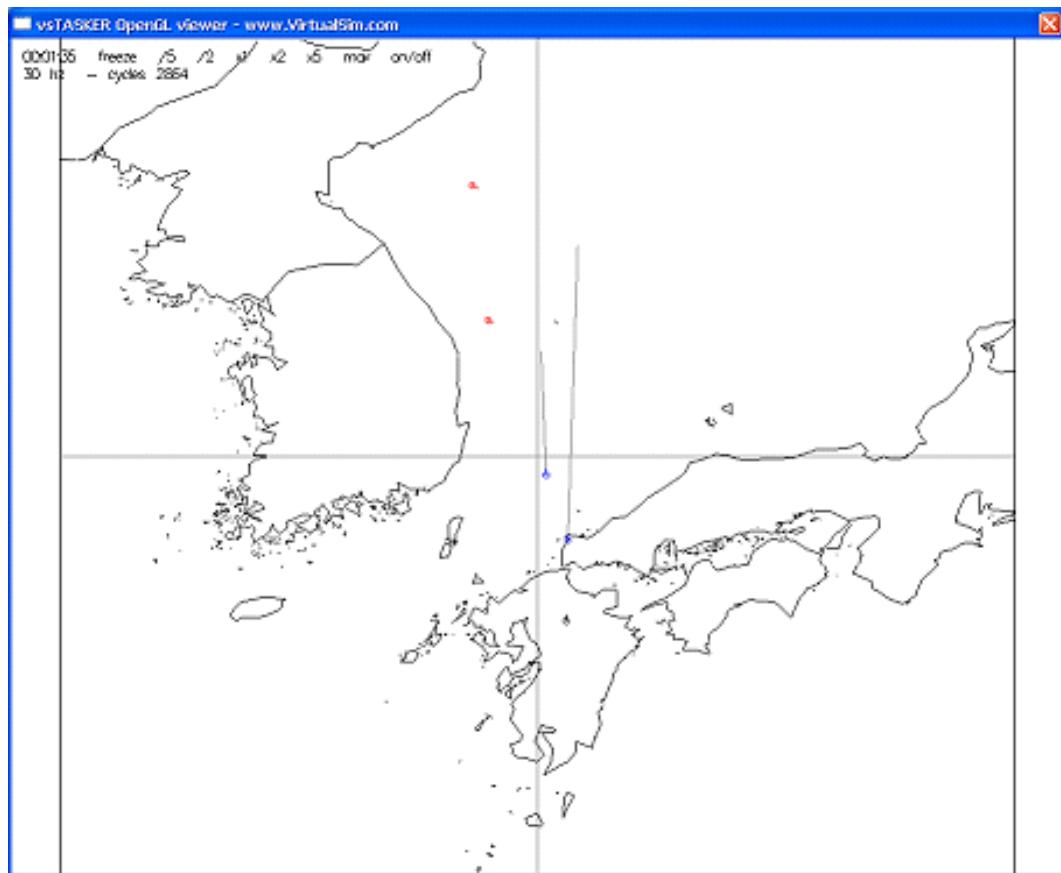
just slide the cursor:  to the right.

Check on the map display the frequency the system can reach. The more CPU power, the fastest the simulation can go without losing accuracy.

A good speed multiplication factor for this demo is **x4**.

Stop the simulation using 

## Ballistic Missiles



On this sample, red ballistic missiles are intercepted by blue missile interceptors. They have their own collision model. The grey line in front of them shows the planned interception point. To avoid missle [sic] the target with high relative speed and angle, the pursuit & collision model put the missile interceptor right behind the target).

# **ISS Orbiting**

Let's open **ISS** sample.

Select  in the main toolbar, then open **iss.db** in **data/db/samples/satellite/iss**



This simulation is showing the current position of the ISS and the current orbit.

This can be cross checked with [www.isstracker.com](http://www.isstracker.com)

If the position is not correct, update the TLE in Navigation Orbits, ISS object. New TLE data can be found at the following address: [www.celestrak.com/NORAD/elements/stations.txt](http://www.celestrak.com/NORAD/elements/stations.txt)

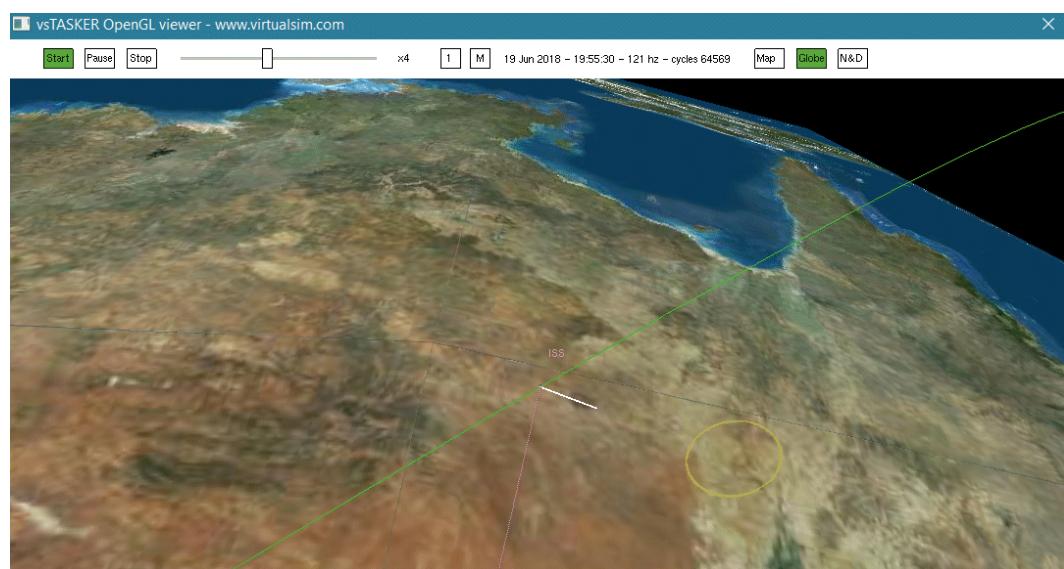
First, open the scenario then set the current UTC time.

Run the simulation.

Open the OpenGL windows then stay on the Map mode or click Globe (wait a little for the earth to be built).

Press N key to focus on the ISS then use the mouse (with left and right button down) to look around. Wheel to zoom.

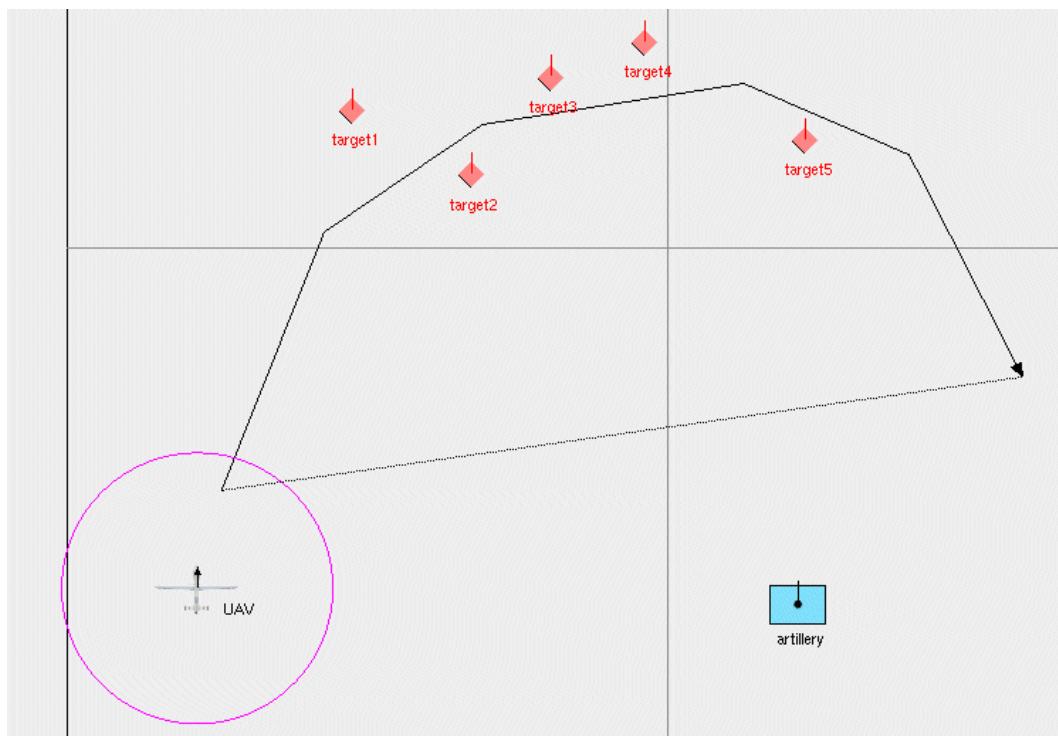
## ISS Orbiting



# Indirect Fire

Let's open [indirect\\_fire](#) sample.

Select  in the main toolbar, then open [indirect\\_fire.db](#) in [data/db/samples/opengl/indirect\\_fire](#)  
Once loaded, your application will looks like below:



In this demo, the UAV follows a trajectory.

It switches ON its sensor at second waypoint and switch it OFF before last.

When it detects a ground foe entity, it sends the coordinates to the artillery which start firing around the position.

When ground targets are illuminated with the radar, they start protecting themselves with a shield, for a random time, as long as illuminated with the UAV radar.

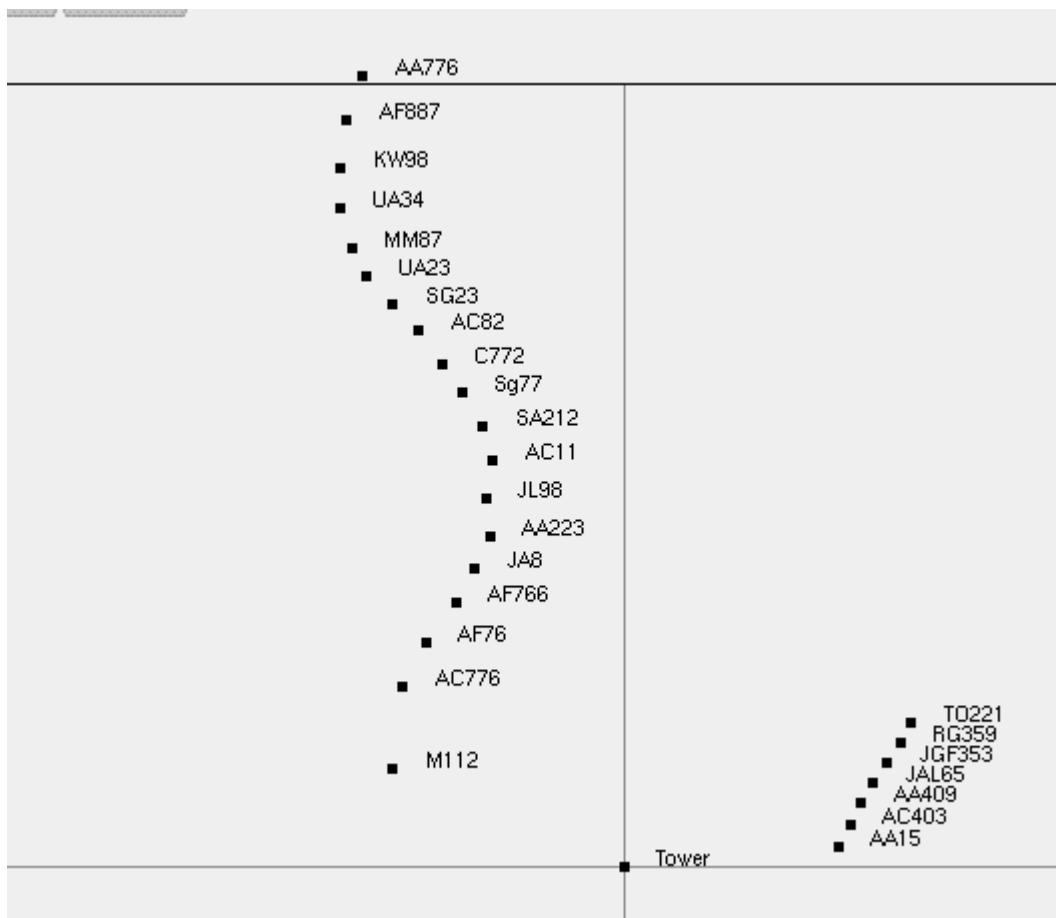
The two targets south of the trajectory are not equipped with the shield. They will be destroyed.

## Air Traffic

# Air Traffic

Let's open **airTraffic** sample.

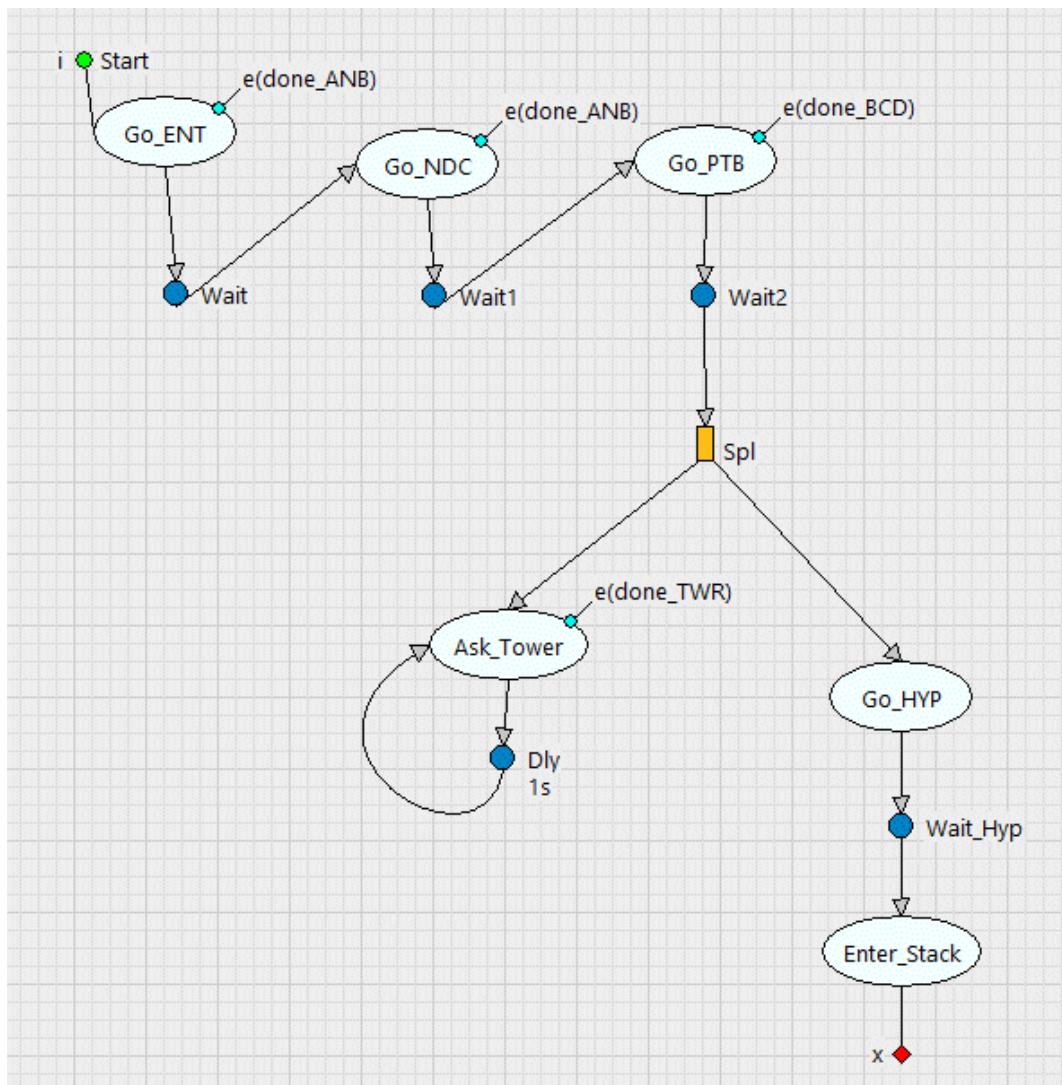
Select  in the main toolbar, then open `msl_ballistic.db` in `data/db/samples/opengl/airtraffic`. Once loaded, your application will look like below:



This simulation combines logic and knowledge in order to manage an air traffic situation where every approaching aircraft request a landing authorization to the tower.

The tower piles up the request and dispatch the aircraft on the various holding patterns. It also manage departing requests. Separation between aircrafts, priorities, taxi way and holding patterns occupation are used by the knowledge algorithms to manage the area.

Simple approach logic with request to land or, if negative, rerouting into an holding stack:



Simple rule example used by a knowledge context to clear or not a specific runway according to various conditions (premises) :

1	If	<code>!scen()-&gt;hasACToward(VDL.pos) &amp;&amp; !scen()-&gt;hasACToward(VDC.pos)</code>
	Then	<code>sFact("RWY14_1_FREE", SCENARIO, this-&gt;db);</code>
Restart Rule After 00:00:01		
2	If	<code>!scen()-&gt;hasACToward(VDL.pos) &amp;&amp; !scen()-&gt;hasACToward(VDC.pos, +90) &amp;&amp; !scen()-&gt;hasACTowar</code>
	Then	<code>sFact("RWY14_2_FREE", SCENARIO, this-&gt;db);</code>
Restart Rule After 00:00:01		



*This demo does not intend to replicate a real TMA controller. The first intention is to demonstrate the combination of Logics and Knowledge as a collaborative approach for solving much more complex simulation cases.*

## Air Traffic

Recompile than start the demo the normal way.

Once the OpenGL windows appear, click on Start button.

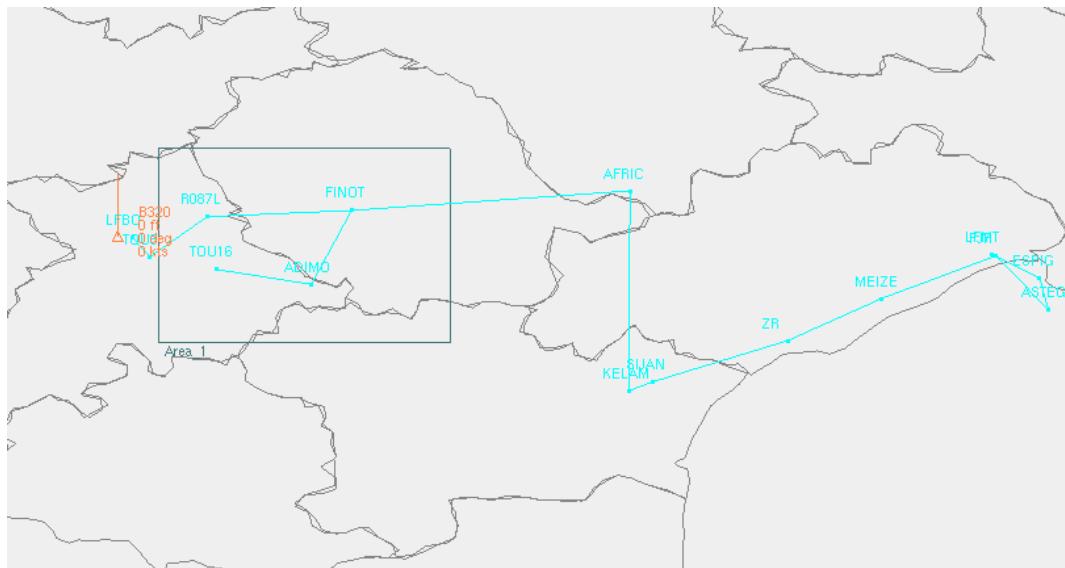
Use the speed slider to adjust the simulation speed.



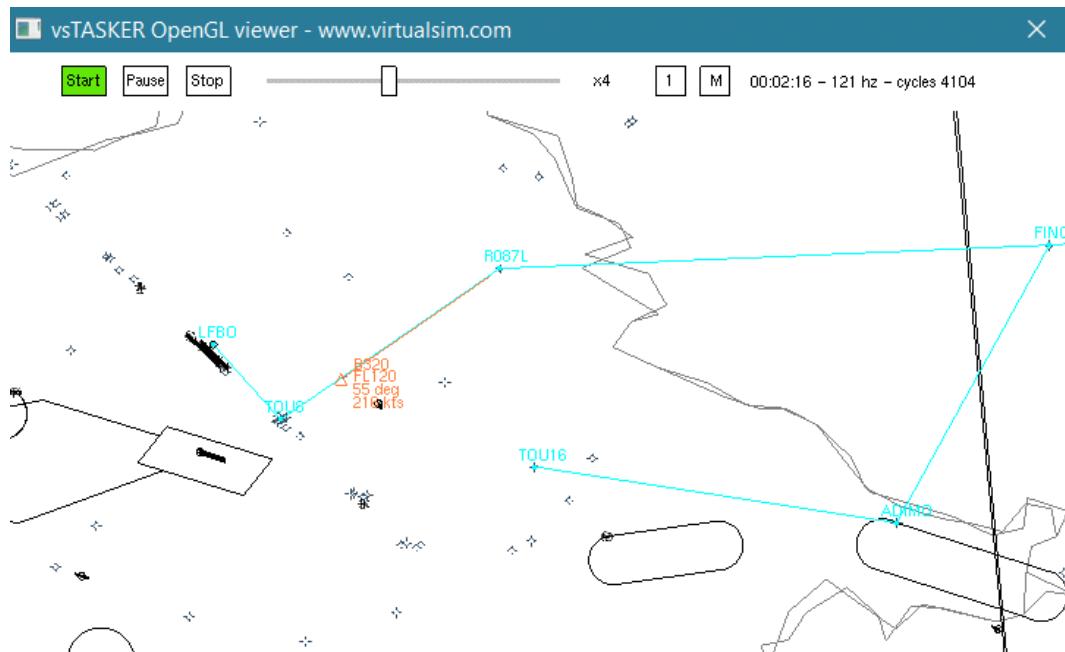
# Alternate Flight-Plan

Let's open **Ifbo-lfrm** sample.

Select  in the main toolbar, then open **Ifbo-lfrm.db** in **data/db/samples/RNAV>Ifbo-lfrm**  
Once loaded, your application will looks like below:

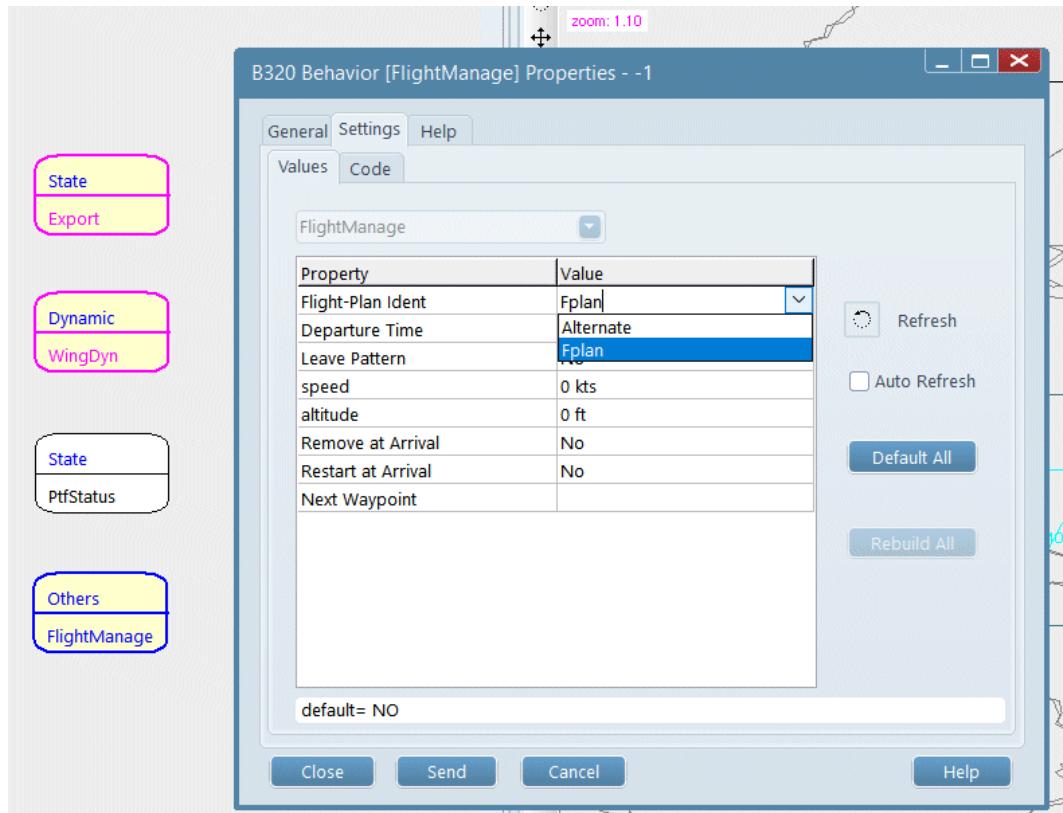


Simple flight plan following with the capability to manually divert using an alternate flight-plan (using the command Alternate) on the B320 aircraft.  
Compile, load and start.

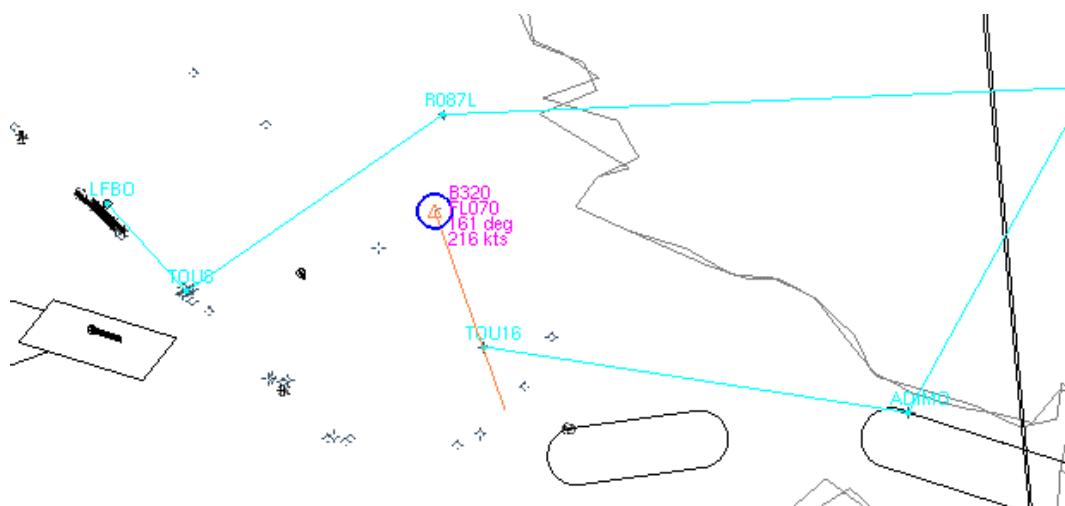


## Alternate Flight-Plan

It may be useful to accelerate the simulation speed to x4 (on the OpenGL window). Once the aircraft is between TOU8 and R087L, select the aircraft, click on Models, open the [FlightManage](#) component, change the flight plan to alternate and press [Send](#).



The aircraft will quit the current flight plan and select the alternate:



## Alternate Flight-Plan



*This demo publishes on the LAN all flight plans, wind, trajectories and zones.  
Useful for another client to retrieve or display this data.*

## **OSG Sample**

# **OSG Sample**

vsTASKER has installed **OpenScenGraph** (OSG version 3.4) so, the demos provided shall work without hassle.

You can open some samples in **data/db/samples/OSG**

# Bouncing Balls

These demos need **OpenSceneGraph** version 2.8 under Visual Studio 2008 or above

You can download it at <http://www.osg.org> although vsTASKER install the correct supported version with release and debug libraries.



Make sure that `$OSG_ROOT` points towards the correct OSG directory.

Make sure that `%OSG_ROOT%\bin; %OSG_ROOT%\bin\osgPlugins-2.8.5` are defined into \$PATH environment variable.

You might need to copy zlib1.dll (into `vsTasker/lib` directory) to `osg_root/bin` directory, in order to load some OSG demos with IVE terrain databases.

Open `mazeball` database in `data/samples/OSG` directory

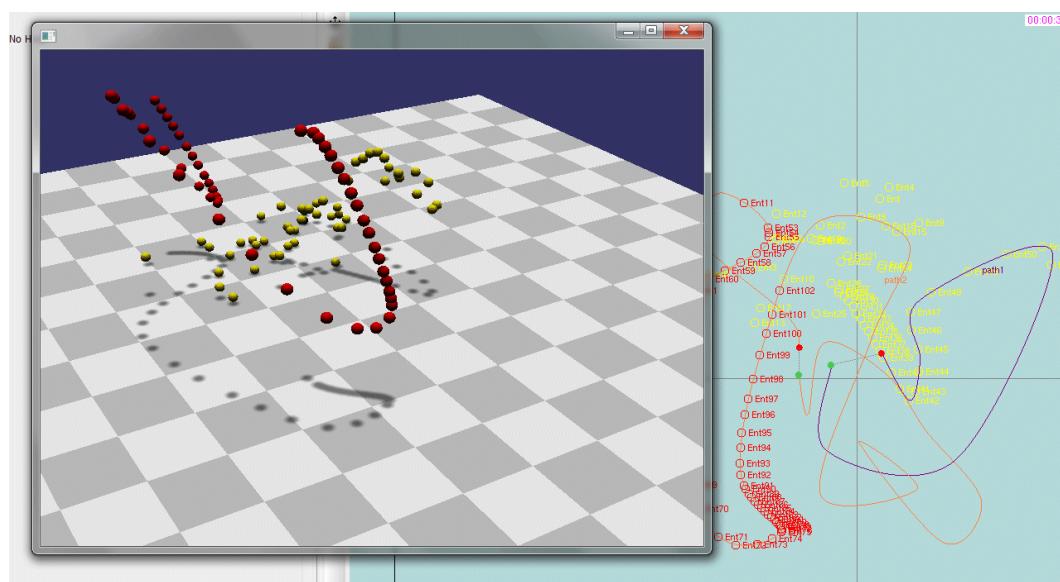
This database holds one simple scenario where balls, on a chess-board area, will bounce on contact. The colored balls always move. The grey balls decelerate and stop.

You can compile the generated code and run the produced exe.

After starting the SIM, use your mouse on the OSG viewer to move the check board (hold down the left mouse button while moving the mouse).

Balls react according to their Model Components. You can change some settings in Bounce and `Stay-Inside` components attached to all balls.

If the visual is slow, you can accelerate it by removing the shadow effect in Viewer OSG, by deselecting Shadows in 3d View.



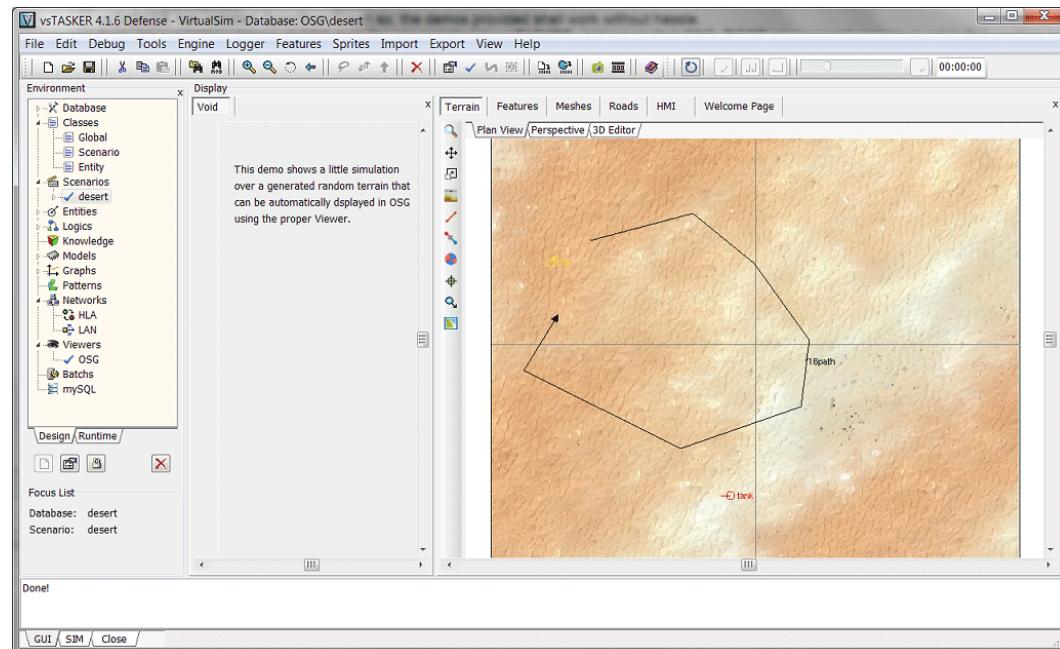
## Bouncing Balls

# Desert Flight

Let's open **desert** sample.

Select  in the main toolbar, then open **desert.db** in **data/db/samples/osg/desert**

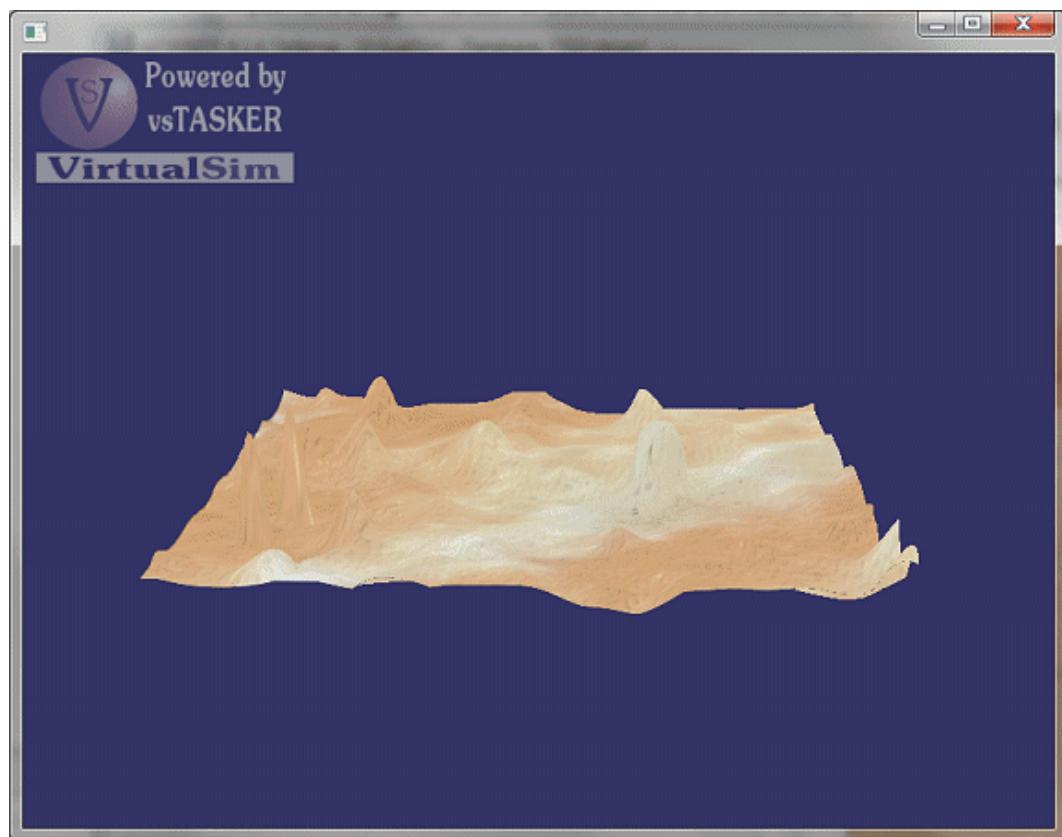
Once loaded, your application will look like below:



Load the simulation engine with the button  . A console application window should pop.

Start the simulation with  , then select the "OSG viewer" application.  
It should look like that:

## Desert Flight



Use your mouse to move the camera around the area.

Hold the left mouse button to move the camera

Use the right mouse button to zoom/unzoom

Select the f16 icon on vsTASKER GUI map.

OSG shall automatically focus on the entity:

## Desert Flight



## Helicopter Control

# Helicopter Control

Let's open **helicontrol** sample.

Select  in the main toolbar, then open **helicontrol.db** in **data/db/samples/osg/helicontrol**

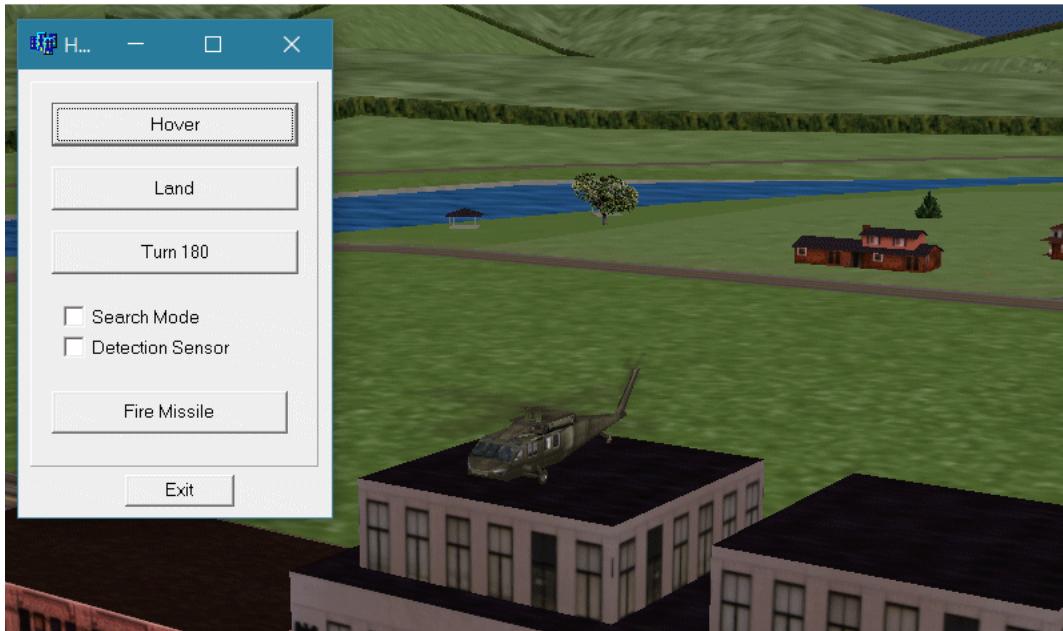
Once loaded, your application will looks like below:



This demo shows the use of user input to control an helicopter and engage user behavior.

Recompile and Start the simulation.  
Two windows should be available:

1. The control panel
2. The OSG view



Using the control panel, press the button "Hover" than "Turn 180".

Engage "Search mode" and "Detection Sensor".

Observe how the helicopter will start looking for the <truck> target.

As soon as the sensor detects something, the helicopter will stop.

Press then "Fire Missile". The helicopter will then move away from the target then face it and fire a missile towards the target.

Press "Land" and see how the helicopter gently land on the ground.

## Suicide Attack

# Suicide Attack

Let's open **suicide\_attack** sample.

Select  in the main toolbar, then open **suicide\_attack.db** in **data/db/samples/osg/suicide\_attack**

Once loaded, your application will look like below:



In this demo, you have one terrorist car which will follow a Plan made of **GoTo** routines until it reaches the last **ExplodeMe** routine.

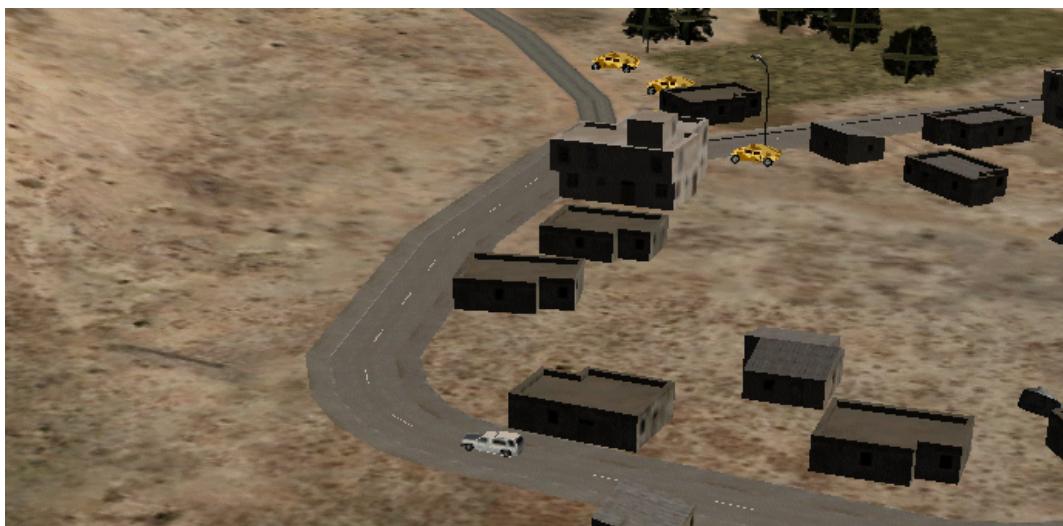
This routine will detonate the **IED** component attached to the car.

The explosion will damage the surrounding entities (hummers)

Compile and run the demo then select on the map the terrorist car and switch to the OSG windows.

Use the mouse horizontally with left button down to move the camera around it and vertically with the right button down to zoom in and out.

## Suicide Attack



The 3D database is a courtesy of **TrianGraphics** and can be retrieved freely [here](#).

## **HLA Samples**

# **HLA Samples**

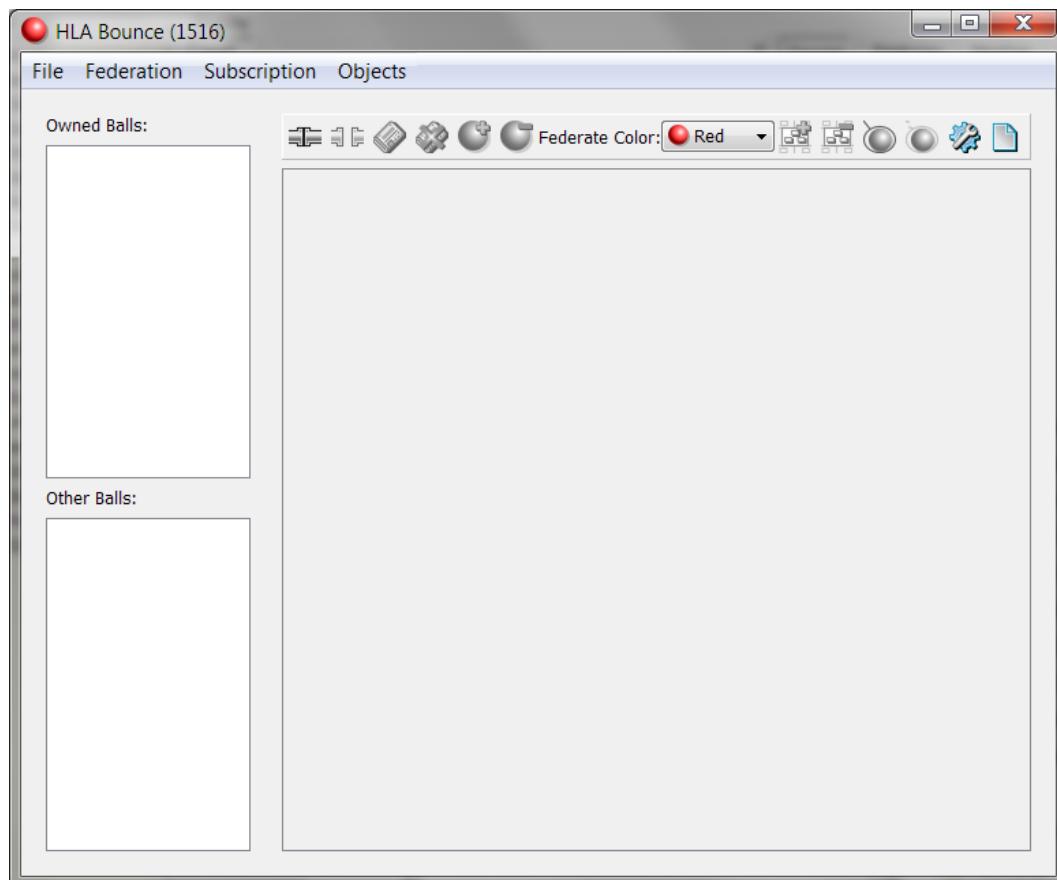
HLA demos are using different RTIs from different vendors. Today, most vendors provides RTI with free evaluation based on 2 federates. This will be enough for our demos. Either we do connect vsTASKER to one provided built-in samples, or we do connect two vsTASKER under a specific Federation.

You can open some samples in [data/db/samples/HLA](#)

## Mäk Bounce

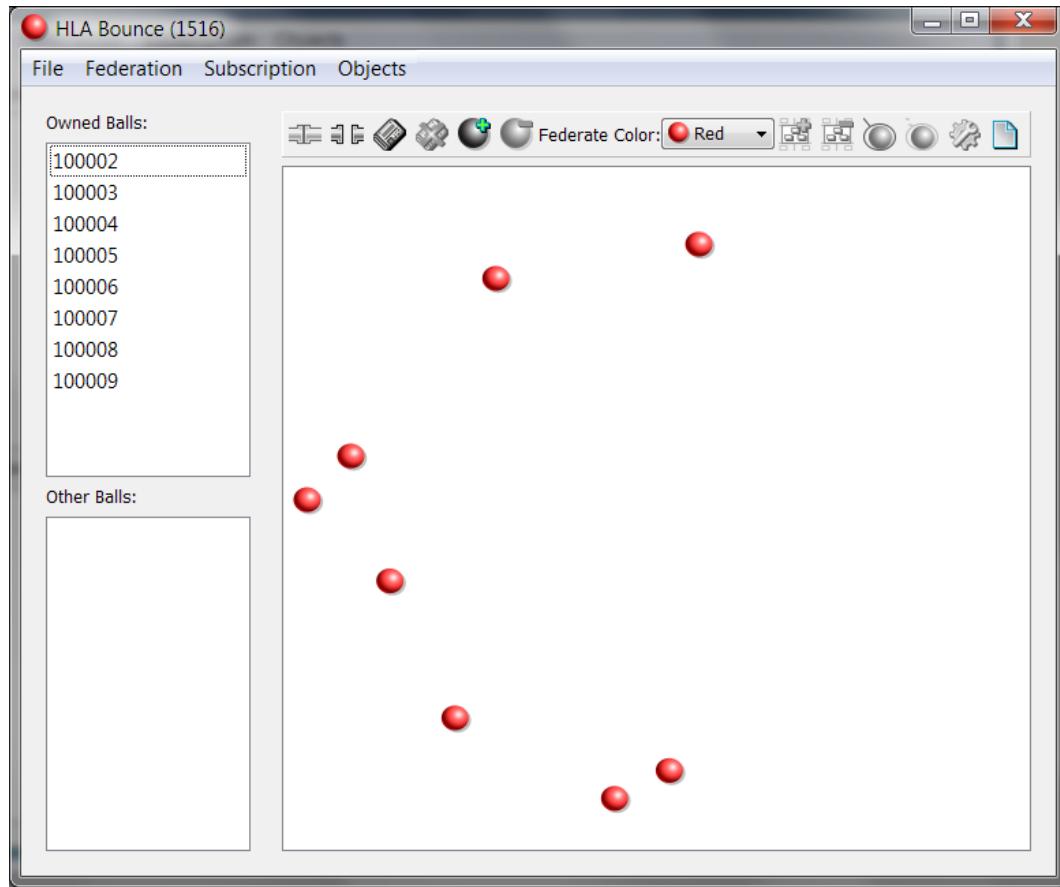
For this demo, we will use the hlaBounce1516.exe sample provided by Mak Technologies on their /bin directory.

The Mak RTI must be installed and purchased from the Mak vendor.  
You can start the sample by double-clicking the executable.



Now, join/create the Federation with and start populating the gaming area with to get:

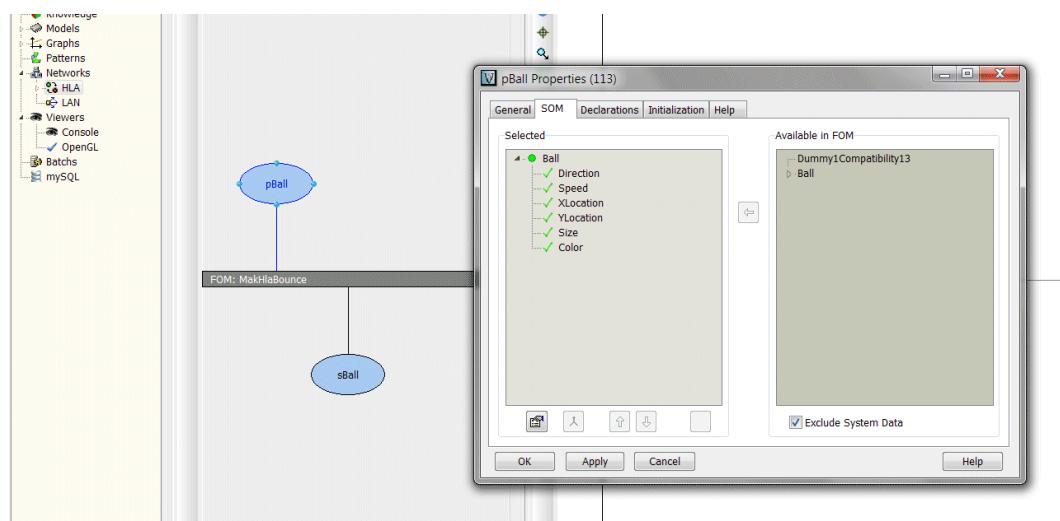
## Mäk Bounce



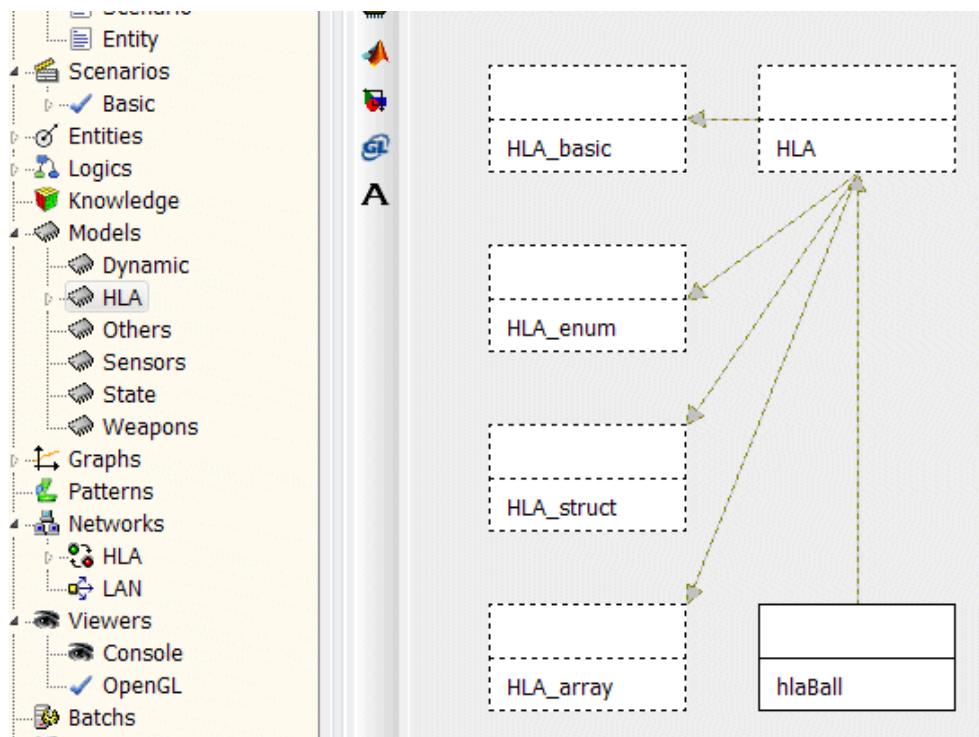
Now, open vsTASKER database Mak/1516/bounce1516 (you can also open the 13/bounce13, both works the same way)

You can see that 2 blue (entity) balls have already been defined.

The HLA part publish and subscribe to the Federation in a usual manner, addressing all Object Attributes:



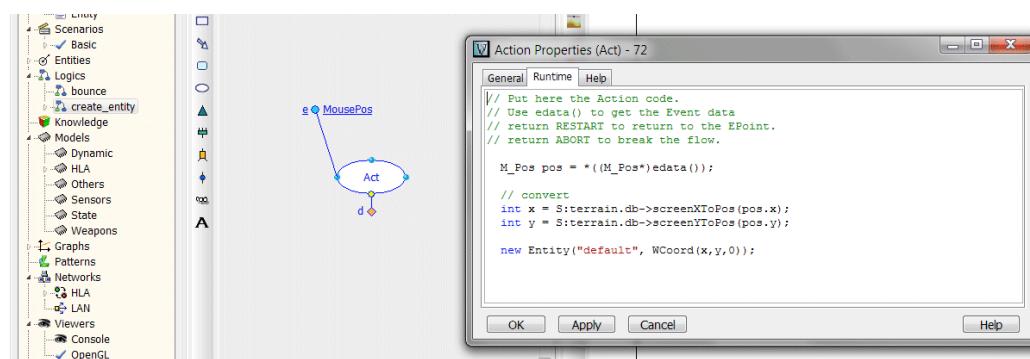
The HLA Models have also been extracted from the Federation definitions and used in both FedItems:



Also, the publisher FedItem allows runtime entities (to be created at runtime)

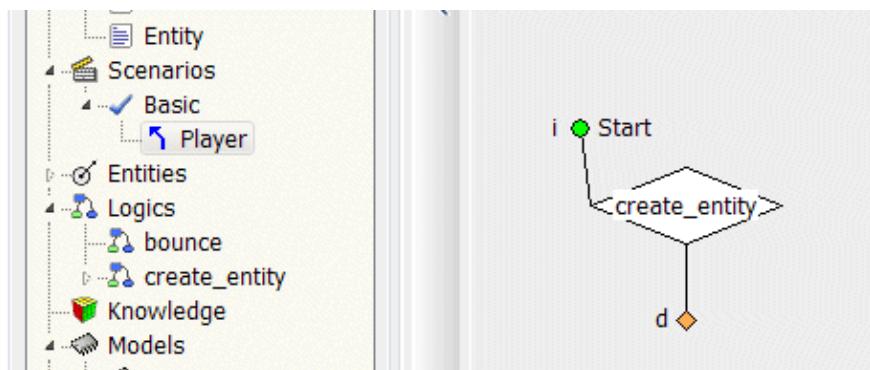
Allow RT Entities

In this demo, we are using an OpenGL window and then, we will react to the mouse click event to create a new (entity) ball at the mouse location:



The `create_entity` logic will be given to the Scenario Player.

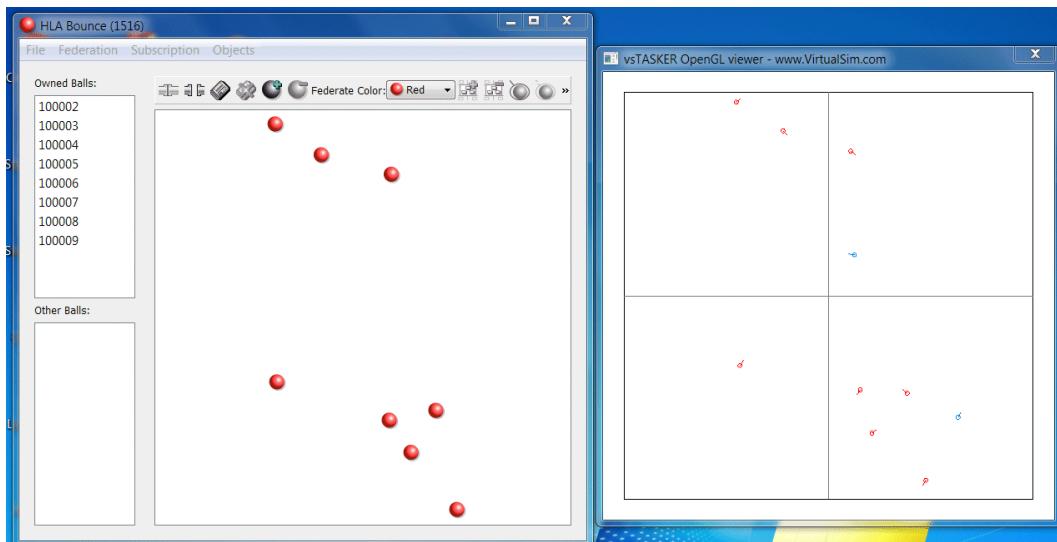
## Mäk Bounce



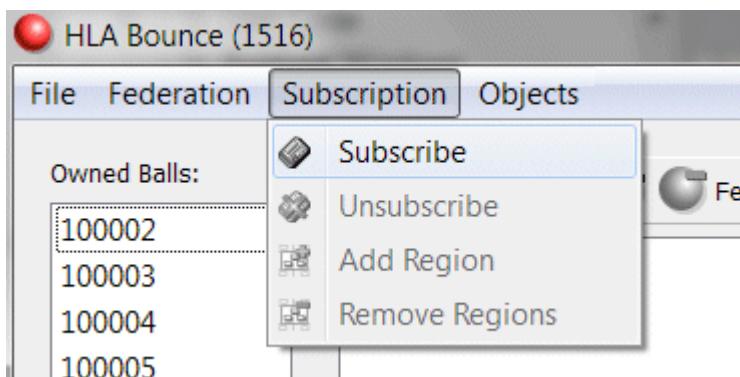
You can now compile and generate the simulation.

At start, you will get an OpenGL window that will connect to the simulation and subscribe to external entities.

All red bounce1516.exe balls will be reflected into vsTASKER OpenGL window:



Now, for the bounce1516.exe to get the published blue (entity) balls from vsTASKER, it is required to subscribe to them:

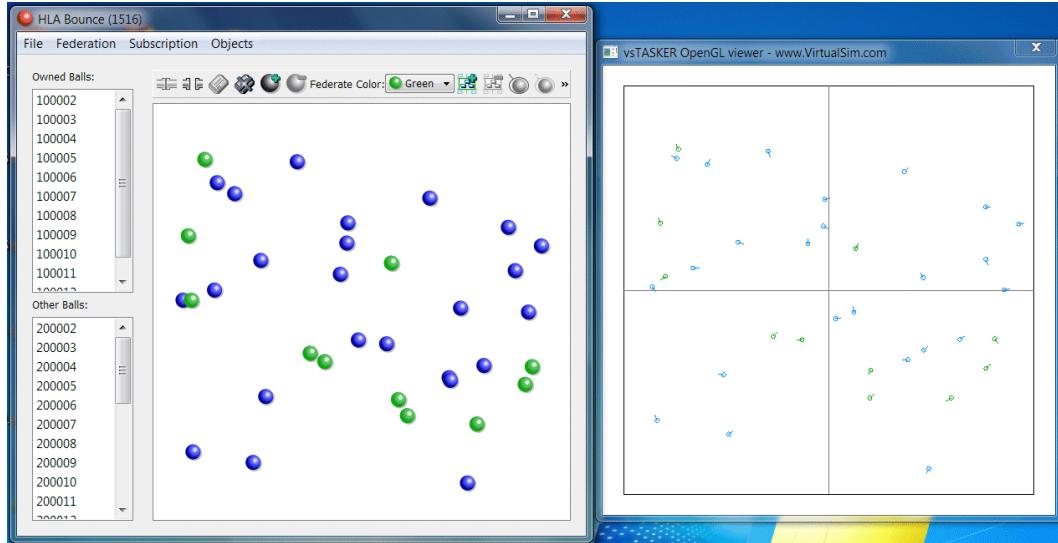


## Mäk Bounce

When Subscribe is activated, all blue vsTASKER balls will be reflected into bounce1516.exe game area.

You can add as many new blue balls into vsTASKER by clicking on the OpenGL window area.

Also, changing the color in bounce1516.exe will be reflected into vsTASKER.



## Pitch RTI Sample

# Pitch RTI Sample

This demo needs **Pitch-RTI** 1516 (version 3.1.9 minimum).



It shows the use of a FOM ([seawar](#)) that contains one only Object ([Platform.Ship](#)) and one Interaction ([Detonation](#)).

One federate ([seawar.exe](#)) publishes the platform ([ship](#)) position and sends detonations using one Interaction. Detonation positions are random. Ship is moving north.

vsTASKER subscribes to [Platform.Ship](#) Object to update in the scenario the *cruiser* platform position. It subscribes also to Interaction detonation to compute the distance between the ship and the detonation. If the damages are high enough to wreck the ship, the entity is disabled and vsTASKER updates published attribute *damages* of [Platform.Munitions](#). [Seawar.exe](#) which subscribes to it stop computations as soon as the ship is wrecked.

In vsTASKER, go to [Distribution::Hla](#) to see how Object federates have been defined.

Above the horizontal bar are Publishers. Below are Subscribers. Shadowed ellipses are Objects, simple ellipses are Interactions.

Each Object/Interaction as well as Attributes/Parameters holds C++ code. Double-click each of them to see their content.

To start the demo, start first pRTI 1615 than load the Simulation Engine. [Seawar.exe](#) will register and start to update [Platform.Ship](#) and trigger [Detonations](#).

Start vsTASKER simulation and click on [Distribution::Hla](#). You will notice that subscribers are all magenta (because receiving data) but publisher is still green (active but not triggered yet).

Go back to the map and see that cruiser platform is moving. Console window is also displaying data.

When the ship is considered destroyed, it will turn gray, publisher will be triggered and seawar.exe federate will start displaying *platform is wrecked!*

# **HMI Sample**

vsTASKER provides an simple editor for creating useful user panels to control the Simulation Engine or whatever system connected to the interface.

These demos show you some simple example of what can be done from within vsTASKER with the HMI builder.



vsTASKER HMI cannot replace major interface builders like Qt Quick Controls or GL-Studio or even VAPS

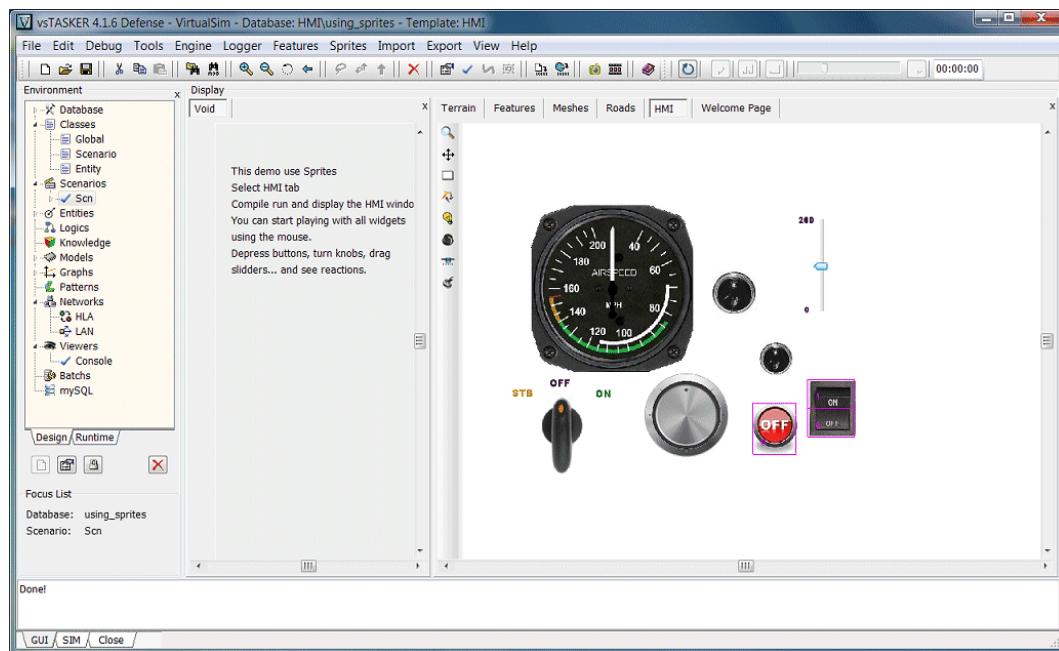
You can open some samples in [data/db/samples/HMI](#)

## Sprites

# Sprites

Let's open [using\\_sprites](#) sample.

Select  in the main toolbar, then open [using\\_sprites](#) in `data/samples/hmi/using_sprites`. Once loaded, your application will look like below:



Load the simulation engine with the button . A console application window should pop.

Start the simulation with , then select the "HMI window" application. It should look like that:



Use your mouse to switch ON/OFF buttons, to rotate knobs or switches and see how lights and the needle is connected to some visual input objects.

See how the right most button shows/hides a sub window. Sub windows are very convenient to create complex user interfaces. There is no limitation in the number of sub windows available in a panel although a sub window cannot host another one.

## Village

# Village

Let's open **village** sample.

Select  in the main toolbar, then open **village** in **data/samples/hmi/village**  
Once loaded, your application will look like below:



In this example, a UAV will follow a search pattern (in yellow) while on the ground, several cars and vehicles will move along the roads.

On the HMI user panel, the player will be able to control the camera payload mounted on the belly of the UAV. By playing with the various sliders, he will be able to rotate the payload and zoom to observe the situation from the sky.

When a moving vehicle appears in the center of the screen, the **Tracking** button will illuminate. It can then be pressed and the payload will automatically follow the vehicle. Once a vehicle is locked, the **Fire** button can be depressed. The locked vehicle will be destroyed from another location.

The UAV can also be controlled manually using the left map. Select the **Goto** button then click on the map. The UAV will quit the search pattern and will head above the mouse location. Press the **Resume** button to go back to the pattern.

## Village



## Integrated Samples

# Integrated Samples

For all of these samples, you normally need a valid license of the third party software to run and modify the VirtualSim provided samples.

Third party libraries or executable are not provided with vsTASKER release version. You will need to recompile the database with the external libraries to get an executable to run.

Contact support ([support@vstasker.com](mailto:support@vstasker.com)) if you have problems running these samples.



*Several of these samples are deprecated and no more maintained. They are kept as archives.*

*They are not available in the Demo version.*

VegaPrime Sample

# VegaPrime Sample

These demos need **VegaPrime** version 3 from Presagis.



## Pendleton

# Pendleton

Open **Pendleton** database in [data/samples/VegaPrime](#) directory. You need to have a game pad (Logitech) attached to your computer in order to control the fighter.

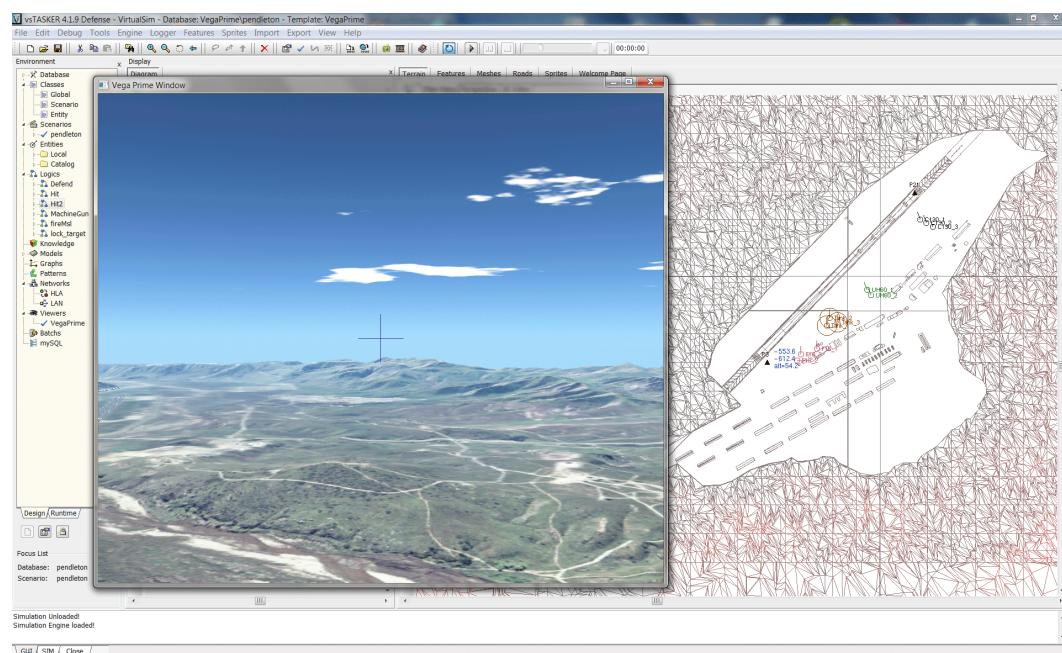
Start the simulation. Once the VegaPrime displays pops up, you are flying the F35. The view is from the cockpit.

Direct yourself towards the airport (slightly on the left). You can fire missile only when the missile tracker is on any detected target (around the blue cross and below 3km distance).

Use the right game pad button to fire missile.

You can fire up to 3 and must wait some time for reload. The main gun is activated with the left button.

As soon as you use the main gun, the three F16 will takeoff and attack you. Exercise is over when all targets have been destroyed.



# Google-Earth Sample

These demos need **Google Earth** (<http://earth.google.com>) and a web server (Apache or IIS).

It shows how to connect and use Google Earth (GE) to display in real-time 3D scene of a vsTASKER scenario.

For installation setup, please refer to the **User Manual**, chapter **General Overview::Viewers::GoogleEarth**



## Takeoff

# Takeoff

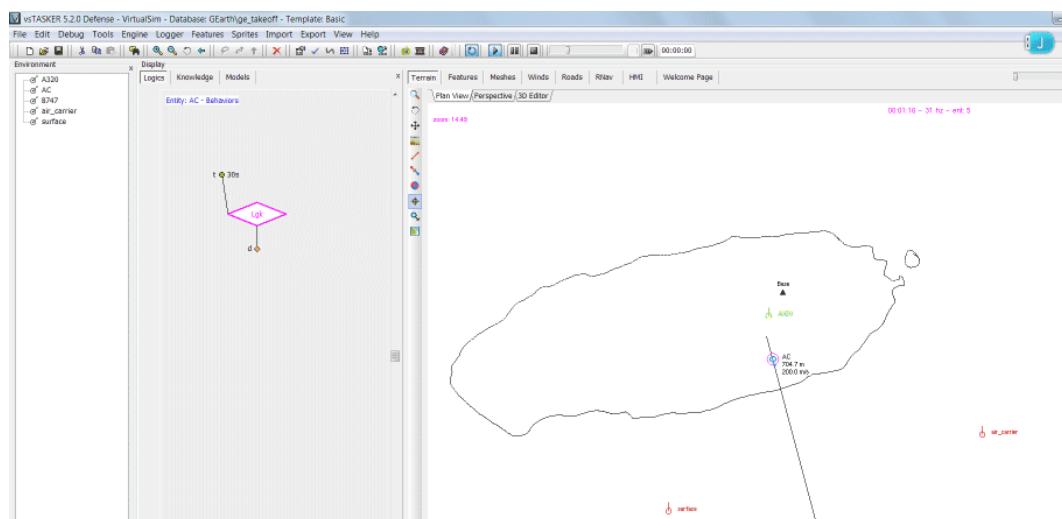
Open [gearth](#) database in `data/samples/integrations/takeoff` directory.

On this database, you will see one aircraft standing on Cheju-do South-Korean Island.

Once set, you can save the database and rebuild or just load the Simulation. GE will normally be automatically called.

Start the simulation and select the Entity. The GE camera will immediately fly to face the Entity on the ground.

You can now navigate around the Entity (red cross) to see the trajectory as defined in vsTASKER and represented on GE.



To embark into the Entity, on vsTASKER, select the Track button . The camera will move to follow the Entity from its position.

**Takeoff**



## VBS-IG Sample

# VBS-IG Sample

These demos are using VBS IG engine from Bohemia Simulation.  
You need to have a licensed version of the software to run the demos



*Specific license is mandatory. Contact VirtualSim if you need the CIGI module.*

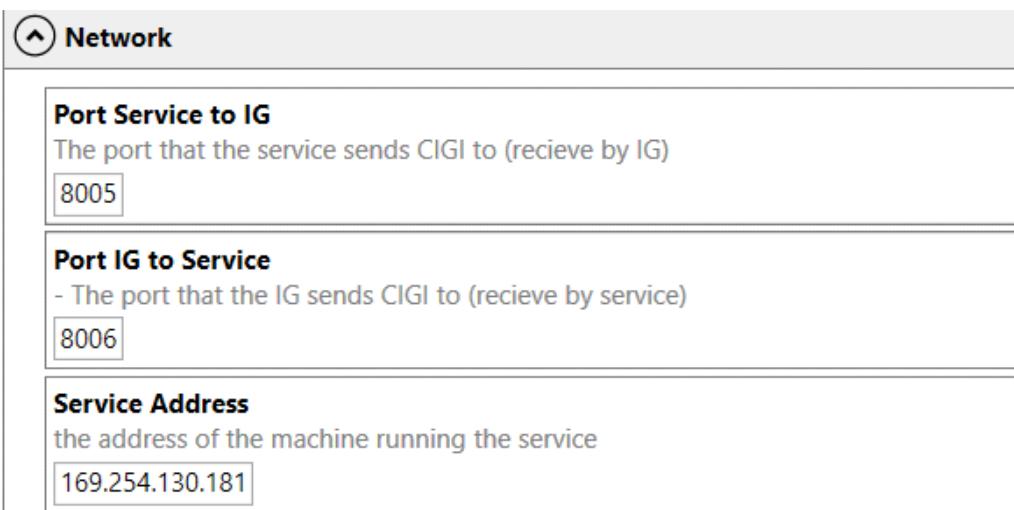
# Anzac

 For this demo, you will need a gamepad. The reason is that VBS IG traps the mouse every time the window has focus and it is very cumbersome to go back to vsTASKER GUI for controlling the simulation. The gamepad is then used for the IG and the mouse for the map. With 2 screens, it is even easier.

Start VBS-IG studio and vsTASKER. Load the [VBS-IG/anzac](#) database.

In VBS-IG Studio, setup the channel 0 and make sure that the ports settings match the one of vsTASKER. In the following example, we suppose both Host and IG are running on the same computer.

Check on both application that they are matching:

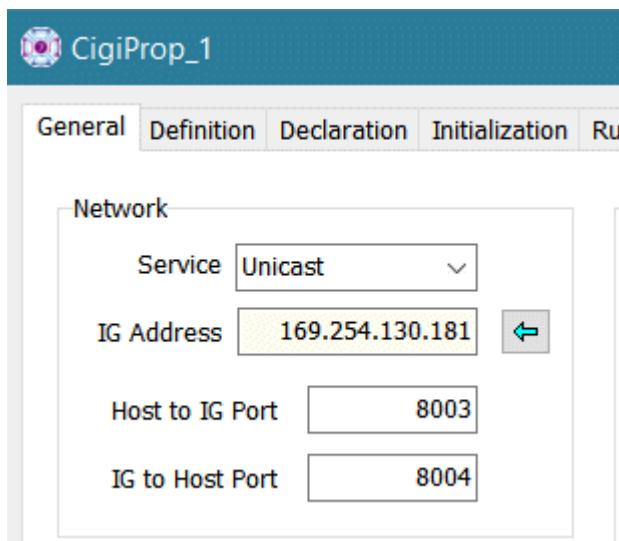


**Network**

**Port Service to IG**  
The port that the service sends CIGI to (recieve by IG)  
8005

**Port IG to Service**  
- The port that the IG sends CIGI to (recieve by service)  
8006

**Service Address**  
the address of the machine running the service  
169.254.130.181



**CigiProp\_1**

General Definition Declaration Initialization Run

**Network**

Service	Unicast
IG Address	169.254.130.181
Host to IG Port	8003
IG to Host Port	8004

## Anzac

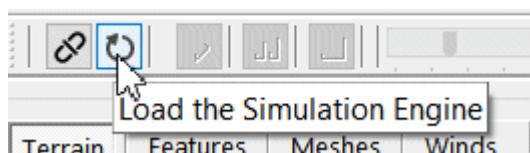
Start the IG from VBS-IG Studio:



and wait for the undersea view:



Now launch vsTASKER simulation engine (you may need to recompile if not available or running) :

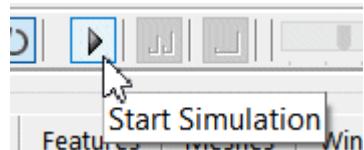


and make sure that the console displays a correct connection to the IG. Normally, you should see VBS-IG loading the Porto database and return to the undersea position (0,0,0)

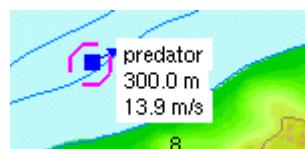
```
=====
=           vsTASKER 7.0.1 (x86)          =
=           =
= Copyright (c) VirtualSim Sarl. 2004-2019. =
= All rights reserved. International Patent. =
=           =
=           Visit www.vstasker.com          =
=           support@virtualsim.com         =
=====

Having loaded environnement for: anzac
Initializing ports to CIGI:
169.254.130.181: Host -> 8003 -> IG -> 8004 -> Host
Successfully connected to CIGI IG server
Force reset IG (Synchronous), load database 1
Waiting for IG Reset...ok
Waiting for IG Operate...ok
Simulation Engine is ready
```

You can start the simulation:

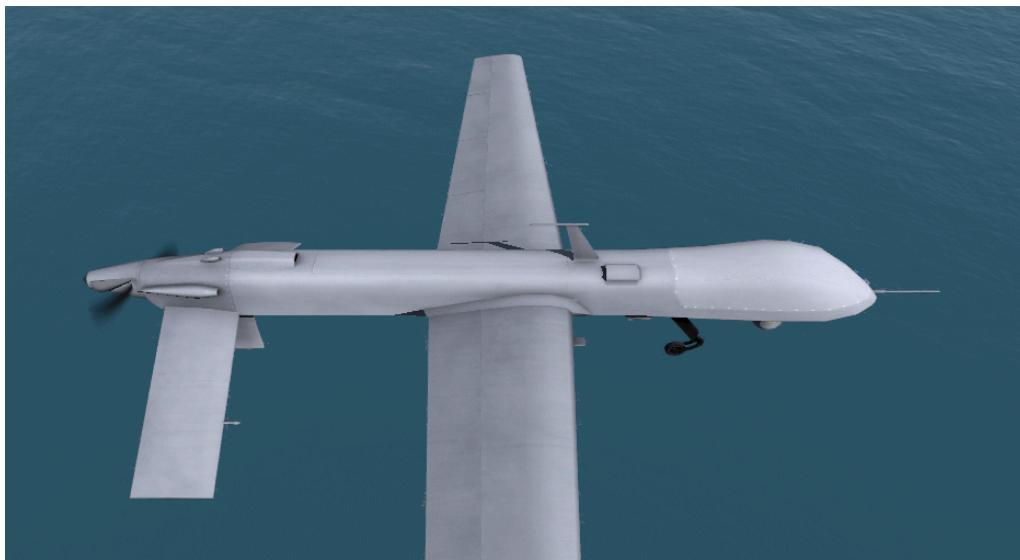


Select the predator entity:



Notice that the IG camera is now focusing on the selected entity:

## Anzac



You can use the gamepad now to move the camera around the focused entity using the left thumb stick.

See [Gamepad Settings](#)

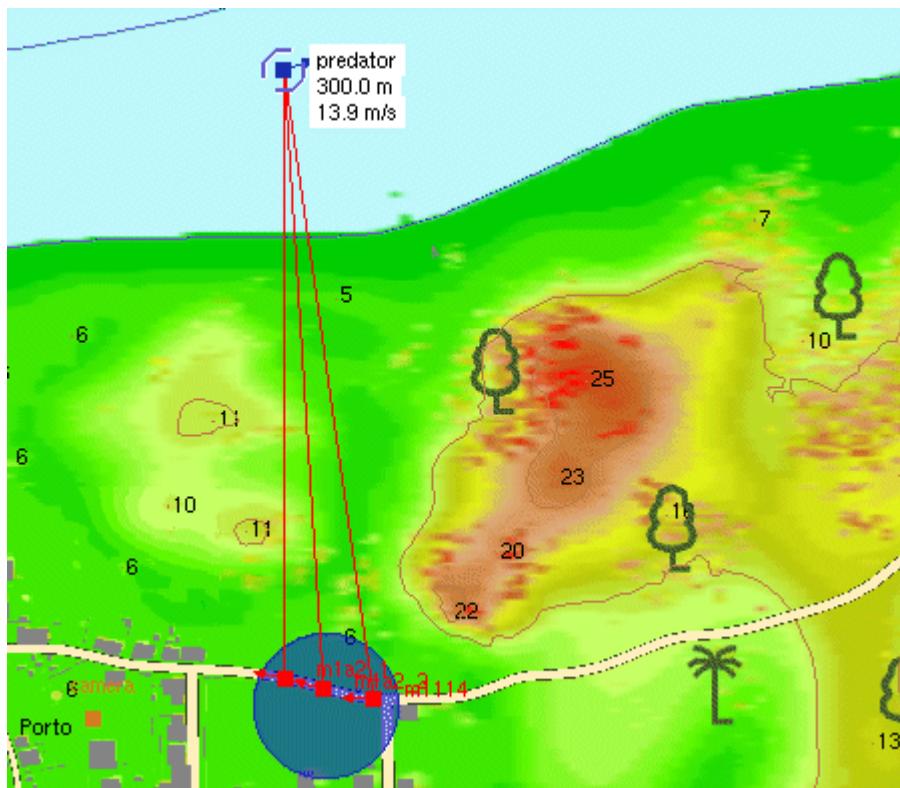
Now, press X (blue) button to activate the Sensor mode (sensor mounted on a gimbal behind the Predator).

You should have this view.



Try to find the convoy of tanks.

When in sight, you can lock it (A green button), unlock (B red button) and most importantly, send the localization to the Anzac ship by pressing the Y (yellow) button. See on the map the data lines.



Switch the focus to the Stinger vehicle:

## Anzac



You can use the right thumb stick to drive it (forward to speed up and left/right to steer).

Use the right button to fire a missile.

# Gamepad Settings



## 1 D-pad Selector



Change the camera focus from one entity to the other: Right one way, Left the reverse way

Up, reset the view (not all demos)

Down, focus to the camera (ghost) entity if available.

## 2 Fixed View



Focus the camera entity (ghost type) of the scenario.

Moving around the camera entity (on the vsTASKER map with the mouse) also moves the camera.

This can be useful to manually explore the terrain database.

## Gamepad Settings

### 3 Camera Control



Moves the camera around the focused entity (or point)

### 4 Right Button



Use here as a trigger to fire when the focus is on the Stinger or any vehicle with firing capabilities

### 5 Right Trigger



When the camera is focused on the Predator or its gimbal sensor, zoom out.

### 6 X (blue)



When the camera is focusing on the Predator, activate the Sensor mode.

### 7 Y (yellow)



When the camera is in sensor mode, select targets and sends the data to the Anzac ship for attack.

8

## B (red)



When the camera is in sensor mode, unlock the gimbal to track the entity centered on the view.

9

## A (green)



When the camera is in sensor mode, lock the gimbal to track the entity centered on the view.

10

## Driving control



When the focus is on a vehicle, use this control to drive it (forward and backward to speed up and down; left and right to steer)  
This stick can also control an articulated part, once selected you can orient it around its two axis.

11

## Left Button



No use.

12

## Left trigger



When the camera is focused on the Predator or its gimbal sensor, zoom in.

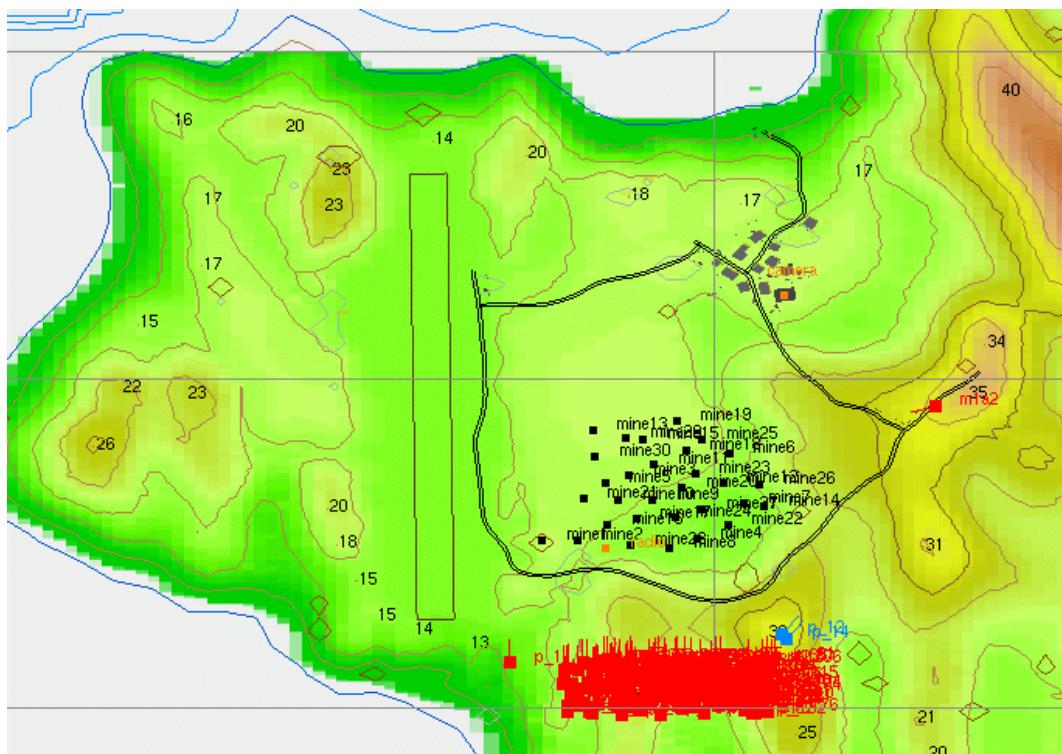
## Infantry

# Infantry



*Run the Anzac demo first to have something running and to learn how to use the gamepad.*

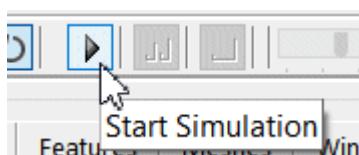
Start vsTASKER and load the [VBS-IG/infantry\\_charge](#) database.



Make sure that the CIGI network settings are consistent between Host and IG.  
Start the IG from VBS-Studio (run the [Anzac](#) demo first to know how to do).

Recompile vsTASKER database then load the simulation. Check that the IG is in standby mode, below the ocean surface.

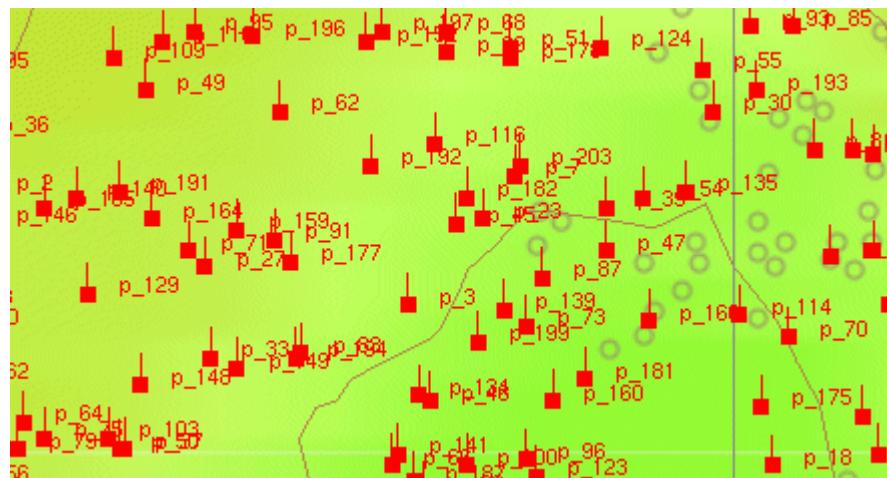
You can start the simulation:



Select **any entity** inside the infantry so that you can have a good view of the infantry.

Press **SPACE** key to activate the focus on the IG (it is not automatic like on the Anzac sample).

## Infantry

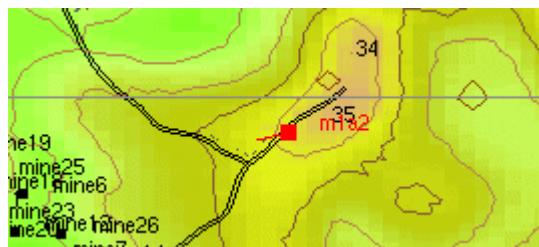


Use the gamepad **left stick** to move around the selected soldier.  
Then use the **A** (green) button to start the attack (see that all soldiers start to move).  
You can let the camera on and see the explosions when they reach the mine field.



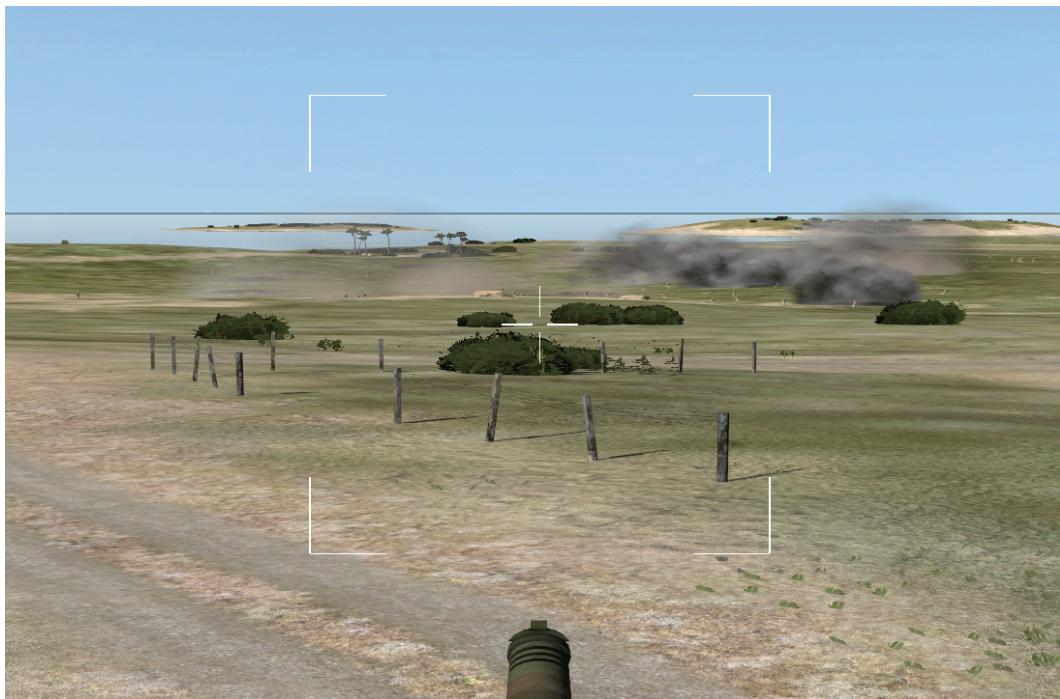
You also can switch to the tank.  
Select the **m1a2** entity on the map then SPACE:

## Infantry



Access the gunner seat by pressing Y (yellow):

## Infantry



Move the **turret** using the **right stick**.

**Zoom** using the **left and right triggers**.

**Fire** the **main gun** using the **right button**

**Fire** the **machine gun** using the **left button**

Hope you enjoyed.

## VBS-3 Sample

# VBS-3 Sample

These demos are using VBS3 game engine from Bohemia Simulation.  
You need to have a licensed version of the software to run the demos

vsTASKER provides a gateway to VBS3 using a LAN communication with a special plugin loaded by VBS.

Go [here](#) to learn more about this integration.



You need to compile the `vsTaskerPlugin.dll` (in `Runtime/VBS-3/plugin` and install it into `VBS3/plugins64` directory prior to run the demo.

# Concept

The integration with VBS is done using a special protocol between the [VBProxy](#) and the [VBSPlugin](#).

Because vsTASKER simulation engine is 32 bits and VBS3 is 64 bits, embedding vsTASKER simulation engine as a DLL for VBS is excluded. But even if both were 64 bits, the solution would not be viable as the DLL is called from VBS at an inconsistent rate and vsTASKER simulation engine needs a steady frame rate to work. Besides, the second limitation is that loading VBS (and the terrain + the VBSPlugin) every time a new change is made in the scenario would take too much time. It is much efficient to have on one computer (or the same one with two screens, preferably, as VBS capture the mouse) running VBS and the plugin and another one with vsTASKER sim.

The protocol between [VBProxy](#) and [VBSPlugin](#) is described in /Runtime/VBS-3/vbs\_interface.h

One [VBProxy](#) component should be used per database and attached to the Scenario Player.

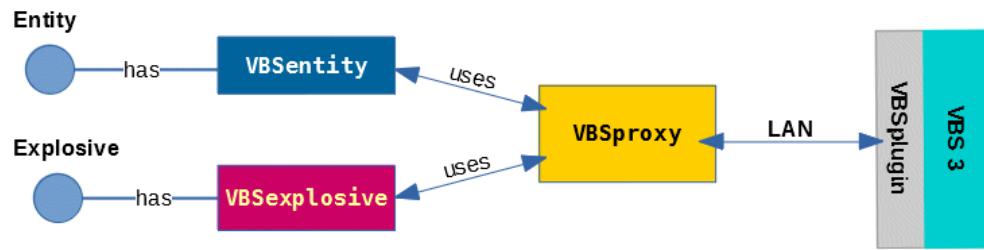
Every entity which needs to exist in VBS must either have the [VBSEntity](#) component (whatever its type) or [VBSexplosive](#) (for mine, IED or any exploding device).

These components (inheriting from [VBScore](#)) are specifically set up for each entity (see the Developer Guide for description of the parameters) and are using [VBProxy](#) for specific commands. [VBProxy](#) contains the list of all entities and explosive in the simulation and automatically sync both sides (position, speed, attitude, state...)

The communication between the [VBProxy](#) and [VBSPlugin](#) is very fast and efficient. The interface between [VBSPlugin](#) and VBS is based on an ASI interface which is slow. The only way to communicate with VBS is using a script command encapsulated into a character string. VBS must parse the command and execute it, the same way as it would do from a script. This is not efficient and when a lot of commands have to be issued, the frame rate on VBS side drops.

VBS also send responses to [VBSPlugin](#) at a low speed. Whenever the request is a LoS, a speed or position of any entity, the response is not as fast as we could expect. So, there is always a time lag between the VBS and vsTASKER situation. The more entities, the bigger the time lag. This has to be taken into account.

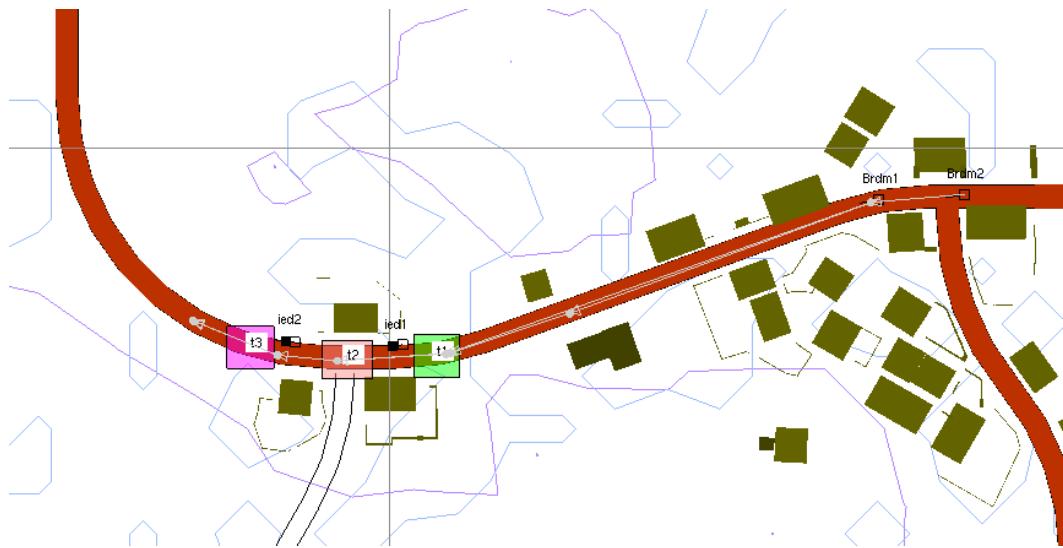
## Concept



# IED

In this demo, we want to test various configuration delays for detonating IED when passing a convoy.

Open the [/VBS-3/ied](#) database:



In the scenario map, we can see the two vehicles (*Brdm1 & Brdm2*) and the two IEDs (*ied1 & ied2*).

The colored rectangles are trigger zones.

Prior to compile and run, select the scenario Player and check the settings of the [VbsProxy](#) component. Values should be as follow:

VBSproxy	
Properties	Values
Ip Address	127.0.0.1
Input Port	19931
Output Port	19930

They are defined in [Runtime/VBS-3/vsTaskerPlugin.cpp](#) in hard (for the moment). If you change these values (and must recompile), do not forget to update the [VbsProxy](#) component settings.

```
#define IN_PORT 9930 // vsTASKER -> VBS
```

## IED

```
#define OUT_PORT 9931 // VBS -> vsTASKER  
#define CLIENT_IP "127.0.0.1"
```

Now, start VBS 3 and load the database. Make sure the plugin is correctly loaded. Start the simulation.

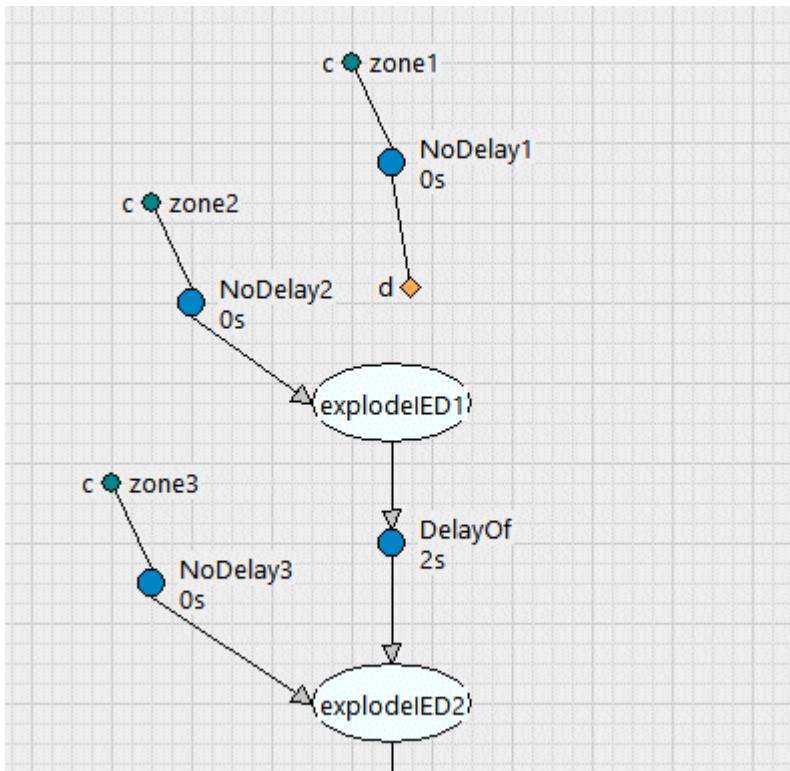
You should see on the vsTASKER map that both vehicles are starting to move along the planned path.

Click on one of them.

On VBS 3 window, the camera should jump to track the selected entity (otherwise, select it on the object list)



When the first or second or even third colored area will be entered by the first vehicle, the attached IED will explode, according to the following logic (belonging to the Player):



You can connect the **NoDelay1** exit point to the action **explodeIED1** (then disconnect **NoDelay2**) or even add some delay time in each **Delay** object and see the result in the damages.

Below, the explosion of the IED1 located in front of a parked car, at proximity of the Brdm:



## **IED**

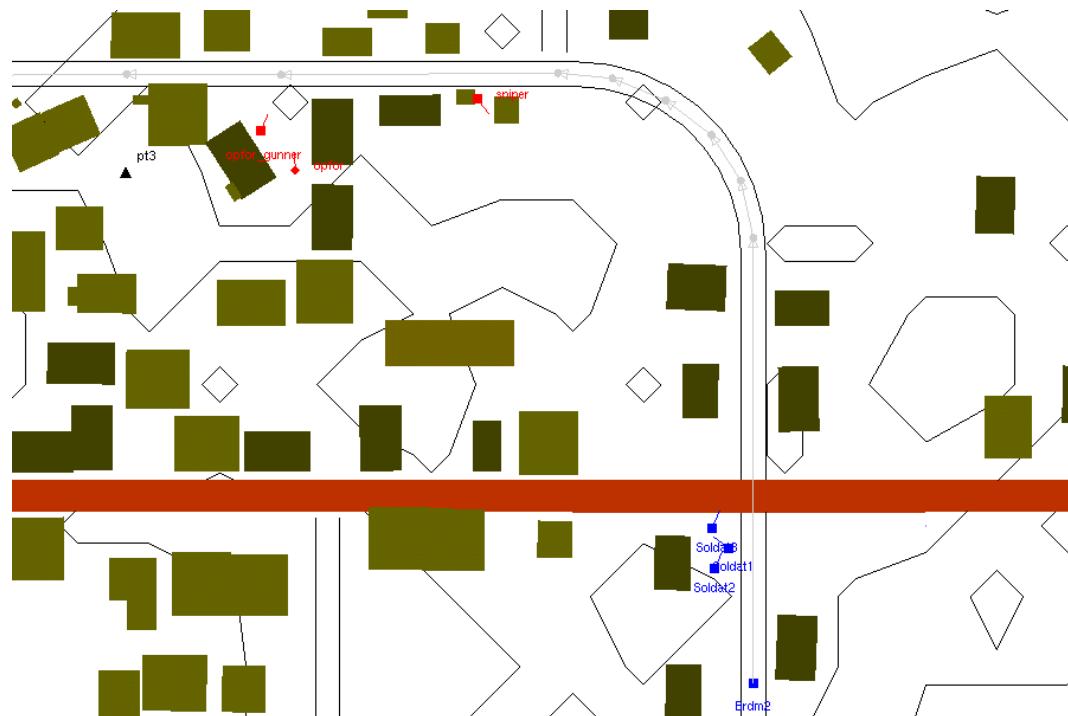
After the explosion of the two IEDs, you may have destruction of both Brdm or just the second one, depending on the delay, speed of each vehicle and lethality of each IED.



# Corazol

In this demo, a Brdm vehicle moving along a road is killed by an opponent tank. Then, a platoon of three soldiers are following behind to engage combat.

Open the [/VBS-3/corazol](#) database:

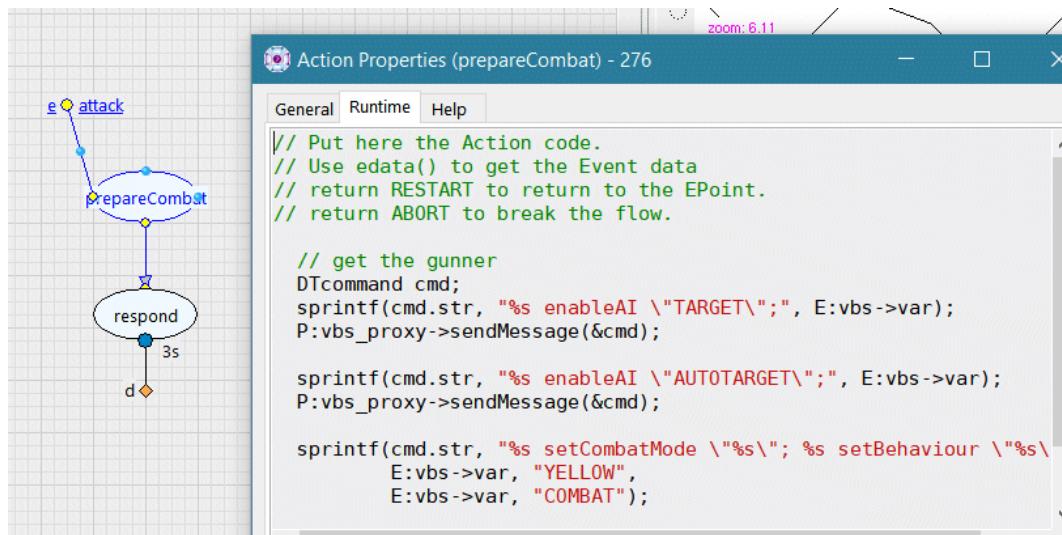


Below, the *Brdm* missed to see the opponent tank (behind):

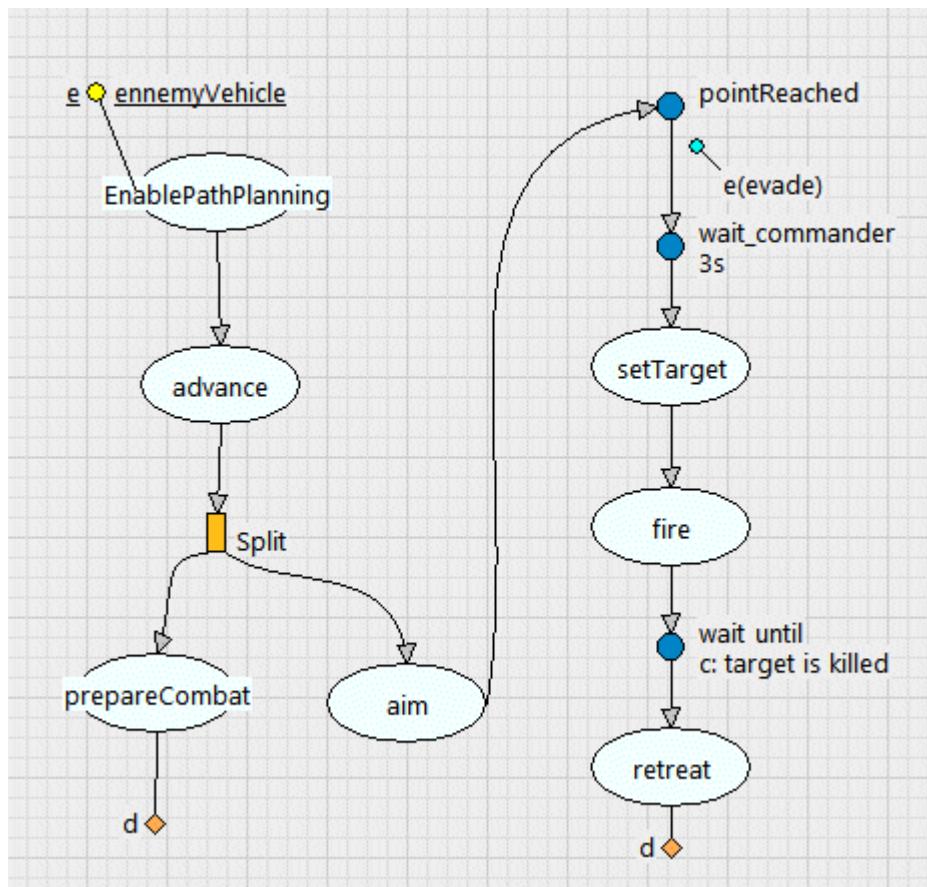
## Corazol



On vsTASKER, the logic given to each entity is based on commands sent to the VBS [ASI plugin](#) using the proxy component:



Here is the opponent tank logic to engage combat when the *Brdm* is detected:



Resulting in the sequence: *setTarget* then *fire*:



Platoon is then moving toward the combat zone:

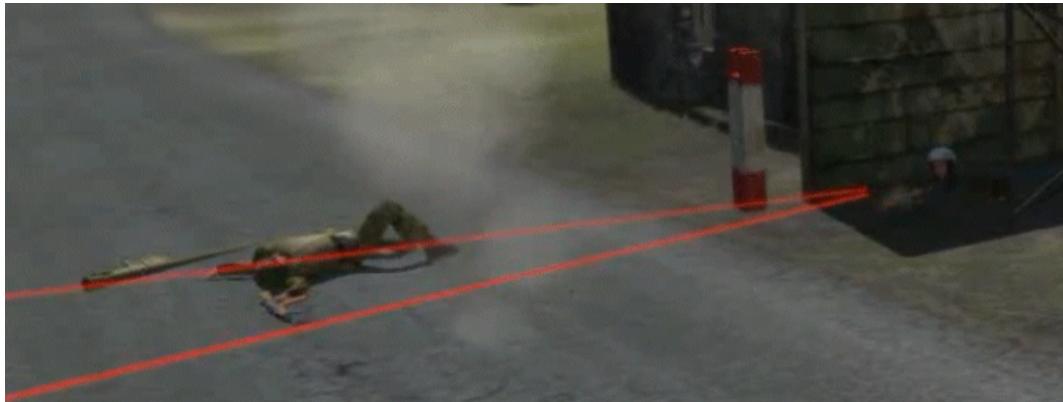
## Corazol



Ready to engage against the opponent tank. See the sniper with the white helmet on the right, in the shadow of the wall:



He will engage fire as soon as the platoon is in position:



## **Titan Sample**

# **Titan Sample**

These demos are using Titan Vanguard game engine from Calytrix.  
You need to have a licensed version of the software to run the demos.

The purpose is to demonstrate the capability of vsTASKER engine to be included into the main Titan Vanguard loop as a DLL host, using directly the C++ API to create and control a scenario at runtime. All the demos has been developed inside vsTASKER, Titan being used as a pure game engine controlled from its programming interface.

# Waiouru

This demo shows a scripted massive infantry attack using the Waiouru database.



*A gamepad is strongly advised with this demo, to change camera target, circle around the tracked entity and fly the helicopter.*

Test demo of a massive scenario including 220 soldiers, 40 mines, 2 gunners and one helicopter.

In this scripted exercise, all soldiers are directed using waypoints and the helicopter is manually flown using a joystick.

`TITAN_DIR` environment variable must be defined and points to `/titan` directory.

VisualStudio 2015 must be used.

Compile and generate the dll (will be created in `/titan/plugins/waiouru.dll`)

You also can copy paste the files `waiouru.dll` and `waiouru.rt` from `/Runtime/Titan/` to `/titan/plugins/`

Open Titan, load the vsTasker `Waiouru` terrain database (copy the `vsTasker_waiouru.tsj` file from `/Runtime/Titan` to `/data/scenarios` of Titan directory. Go to **Scenario Lobby**, open the `vsTasker_waiouru` scenario and enter it.

In vsTASKER GUI, start the simulation and see that entities are created on Titan side. On vsTASKER map, click on any entity to attach the camera;

## Waiouru



Use the gamepad cross button to move the camera around the tracked entity:



Click on the **Leopard** helicopter and start flying it using the gamepad thumbsticks:



Change the **camera mode** using gamepad button **X** and the **sensor mode** using button **A**.

Change **weapon** using button **B**.

**Fire** using the **Right** button.

**Qt Sample**

# Qt Sample

This sample needs **Qt** version 4.8.4 or above.

# Train Doors

Open [train\\_doors](#) database in [data/db/qt](#).

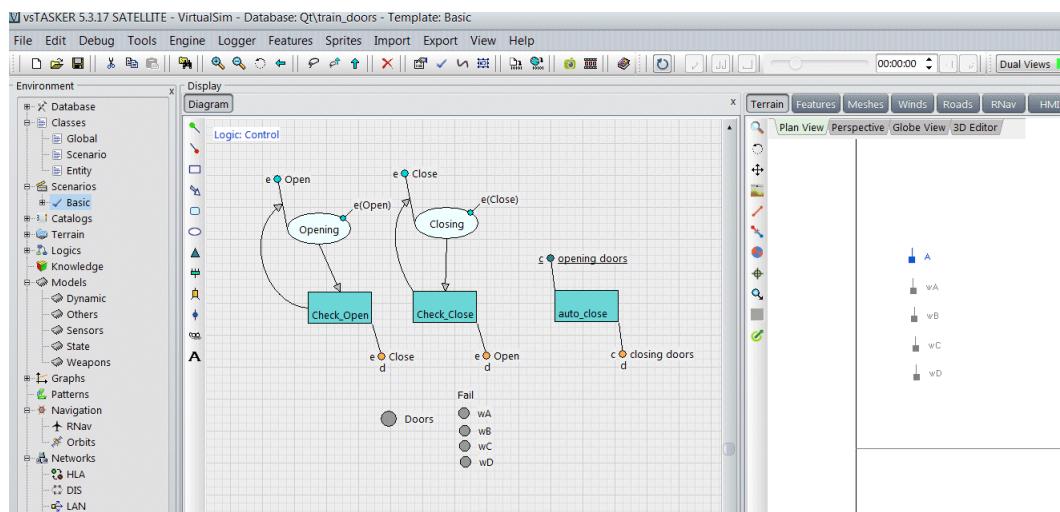
In this sample, a train of 5 cars is defined. Simulation will only cover the door open/close logic.

Select the **car A** and visualize the attached logic like below.

[Open](#) and [Close](#) events are triggered from the Qt panel.

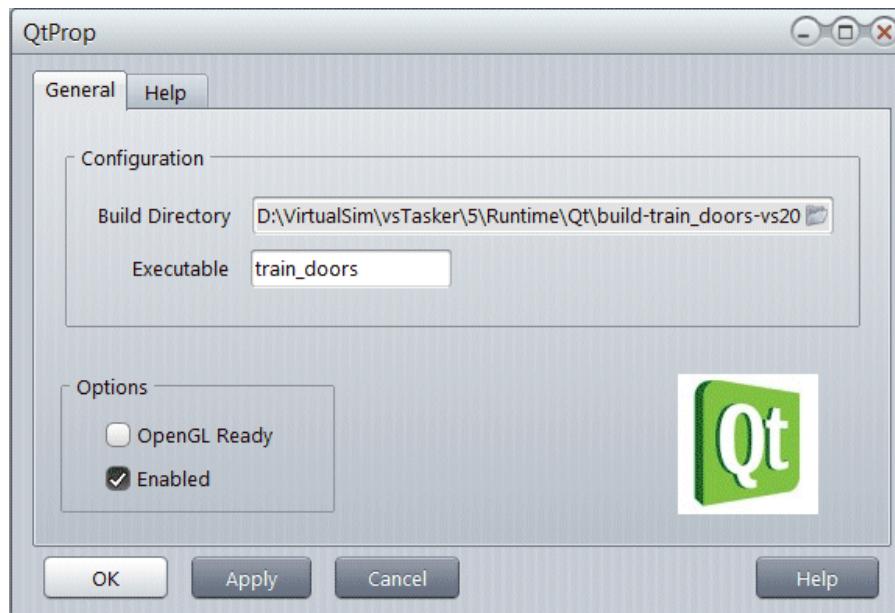
[Check\\_Open](#) and [Check\\_Close](#) are logic tasks that will initiate the opening or closing of all car doors.

Malfunctions can occur randomly. Watchs objects ([wA](#), [wB](#)...) illuminate accordingly.



The viewer is defined to be Qt and the directory where Qt-Creator will generate the executable must be specified in the Build Directory field if vsTASKER is used to automatically recompile the Qt executable. This is not mandatory as Qt-Creator (or Visual-Studio) can also be used for this.

## Train Doors

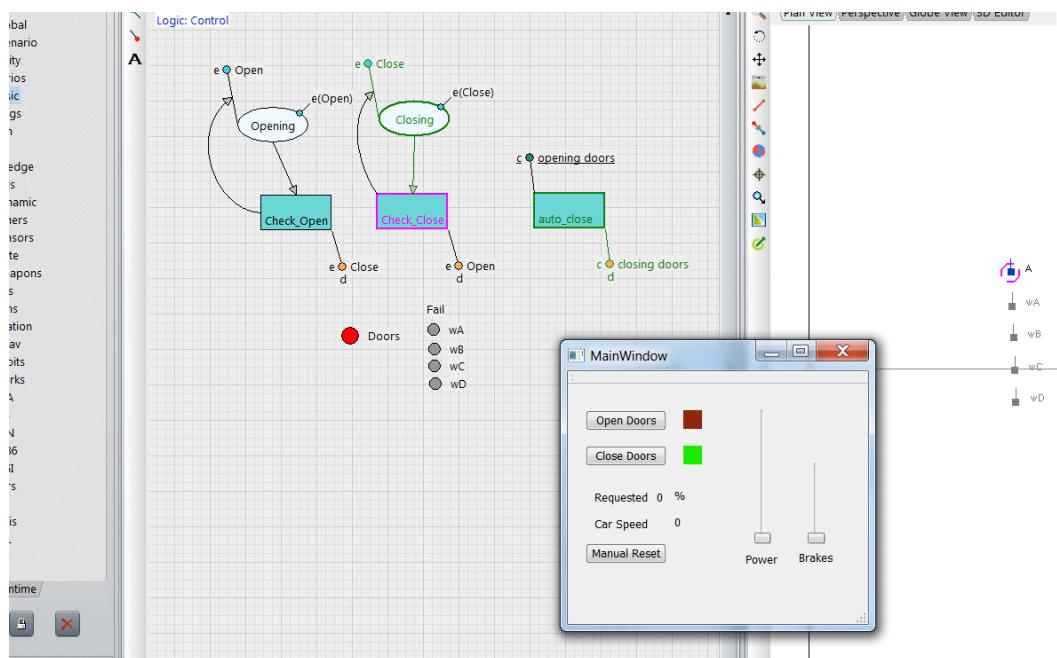


The `.pro` can be opened using Qt-Creator. It is located in `/runtime/Qt/train_doors`. The project includes the vsTASKER generated files. Callbacks are attached to buttons (slots) to trigger the events.

From vsTASKER, you can generate the code.

From Qt-Creator, you can recompile and run the simulation.

At this point, the Qt simulation is standalone. As the shared-memory is created and used by the simulation engine, vsTASKER GUI will monitor the logic in real-time. User can then play with the Qt panel to open/close doors or start/stop the train.



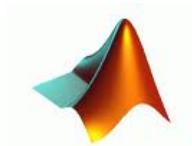
## Train Doors

**Matlab Sample**

# Matlab Sample

This demo has been tested with **MATLAB R2008b**.

Make sure that MATLAB environment variable is set to [R2008b](#) directory.



# Data Plotting

Load the scenario **Matlab/test\_matlab**

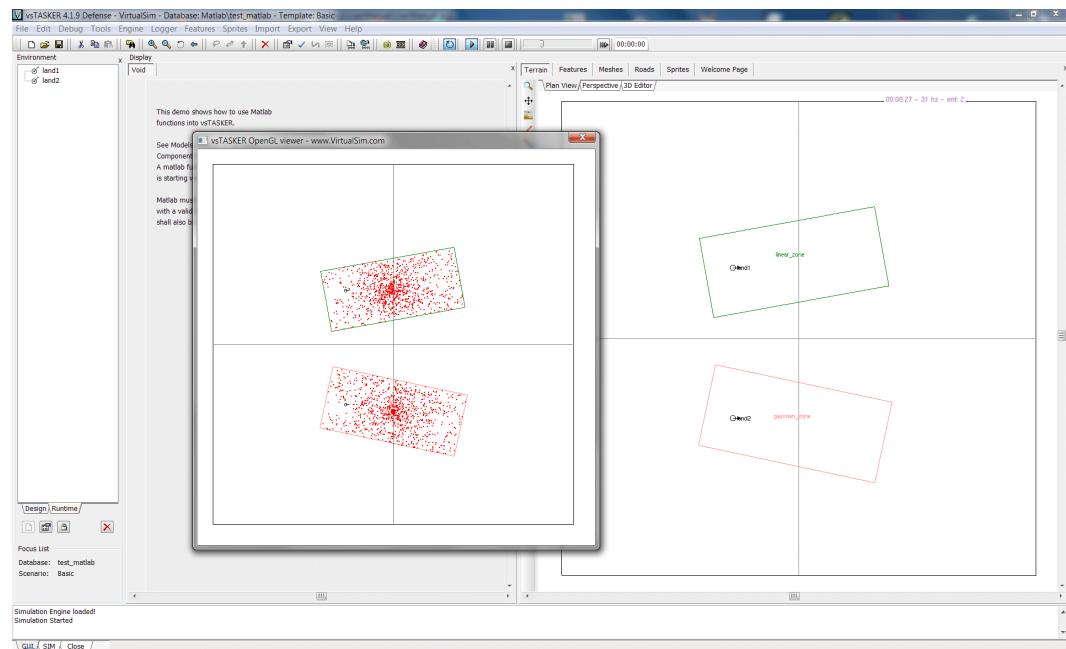
All M-File functions are located in **Models/Matlab**

Start the simulation.

An OpenGL window will pop-up after 5-10 seconds and will show the progression of the two entities inside Special-Zones populated with MATLAB functions.

At the end, a MATLAB Plot window will pop-up.

Make sure you have a MCC compiler associated with MATLAB.



## Simulink Sample

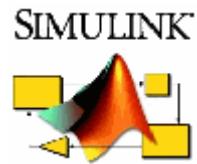
# Simulink Sample

This demo has been tested with **MATLAB R2008b**.

Make sure that [MATLAB](#) environment variable is set to R2008b directory.

Load the scenario [Matlab/test\\_simulink](#)

The model is located in [Models/Simulink](#)



Start the simulation.

A vsTASKER plot window will popup and display the value output by the model.

To change the model, open the model located in [Runtime/Simulink/Model](#)

Generate the code using the RTW. vsTASKER will automatically compile and link the generated code with the Simulation Engine.

Make sure you have **Real-Time Workbench** associated with **Simulink**.

# **STK Samples**

These samples show the different ways for controlling/using **STK** from/  
with vsTASKER.

A valid STK (version 9 or above) license is mandatory for the samples  
to work.



## Communications

# Communications

In this example we have an aircraft type Lockheed EP-3 Orient that is in charge of surveying a maritime area.

It is equipped with an optical rectangular sensor (10x5 degrees aperture) for threat detection. Any ship is detected as a threat. Then, the Aircraft tries to contact the command center to know with answer to apply against the threat.



Because the communication link between the Aircraft and the command center is not good enough, the Aircraft must wait until it gets a clear signal.

Then, when it gets a go, it fires a anti-ship Harpoon like missile.

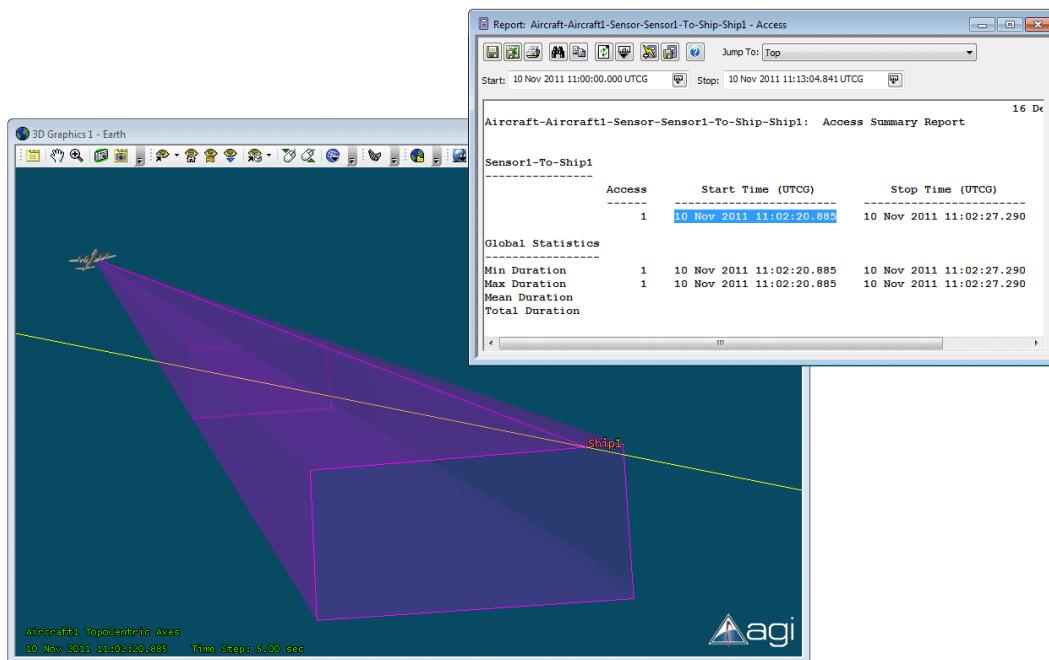
STK is used for its analysis capabilities:

- Build of a trajectory for the Aircraft with the Aircraft Mission Modeler tool (STK/AMM)
- Build of a great arc trajectory for the Ship1
- Setting of the optical and electronic components
- Line of sight computations between the optical sensor and the ship
- Computation of the signal quality between the aircraft emitter and the command center receiver

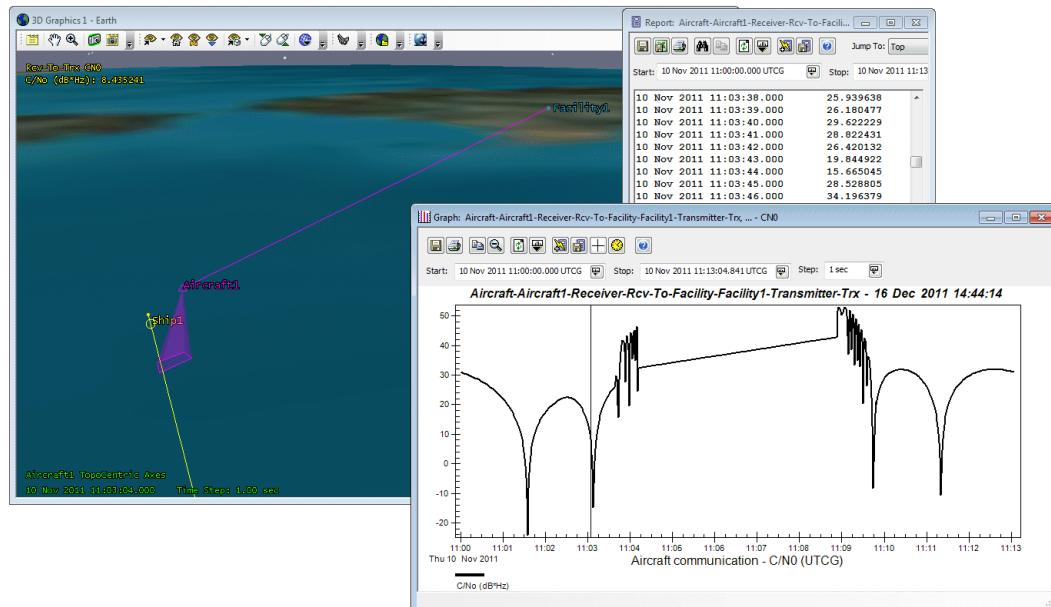
vsTASKER is used for the logic design, behaviors modeling and the missile capabilities:

- Sensor models that remotely used the STK one for the detection events
- Emitter/Receiver models that use STK counterpart for the signal quality report
- Missile component
- Logic to react to the threat detection (Aircraft to Command Center, Command Center to Aircraft, Aircraft to Missile, etc.)

## Communications

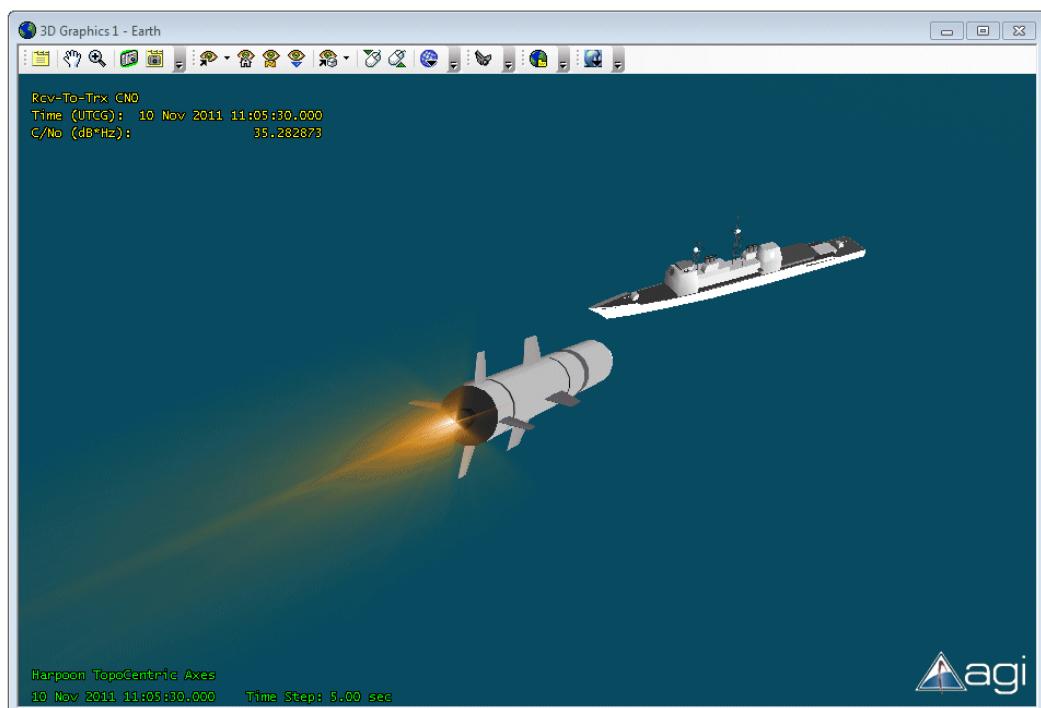


Here, a threat is detected at: « 10 Nov 2011 11:02:20.885 »



The Aircraft communication signal is bad (less than 30 dB.Hz until « 10 Nov 2011 11:03:46.000 UTCG »)

## Communications



The missile is controlled by vsTASKER and visualized into STK.

# Data Provider

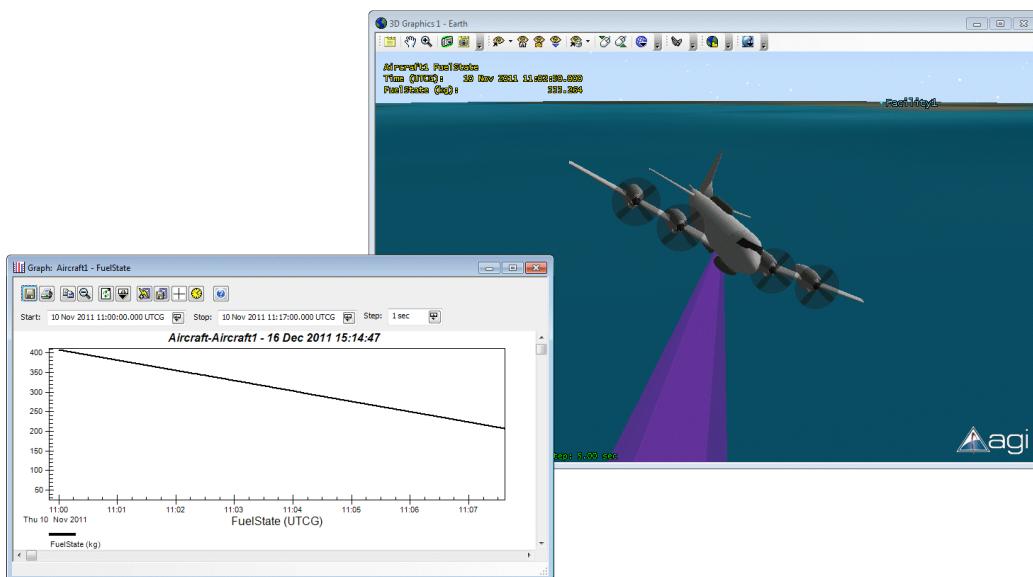
This scenario is almost the same as the Communication sample. The difference is that the Aircraft continue the survey mission after the Missile has been fire.

After a certain time, low fuel level will force abortion of the mission. Another Aircraft must then be sent to replace the first one and continue the mission.



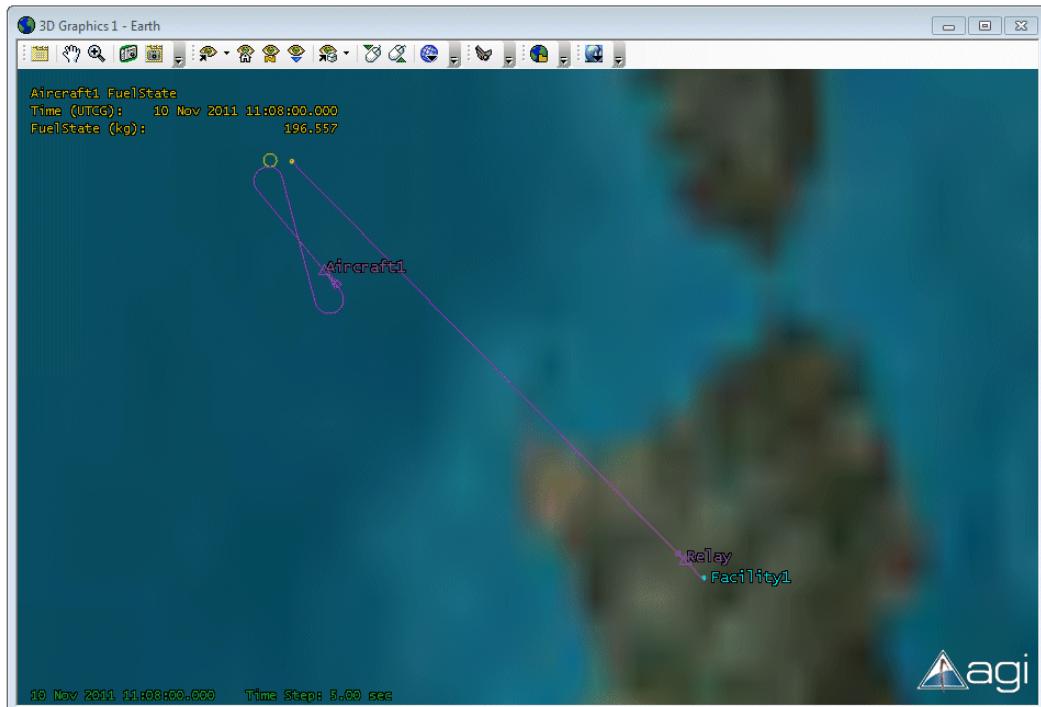
STK is used for the scenario modeling and the analysis capabilities: Fuel level is computer by STK using the Mission Modeler.

vsTASKER is used to trigger logics and behaviors according to STK runtime data:  
Component to check the fuel level of the STK Aircraft  
Trajectory runtime modeling of the second aircraft that will support the first one.  
The second aircraft will be handler by vsTASKER (not using the STK AMM)  
Logic to react to the Low fuel event of the first Aircraft.



Low fuel level (less than 300 kg) at time « 10 Nov 2011 11:04:06.000 UTCG »)

## Data Provider



The second aircraft (relay) is heading towards Aircraft1 to take over.

## Active/X STK Viewer

This demo needs **STK** 9 (8 minimum) from AGI to be installed.

Open **bpatrol** in **data/samples/STK** directory and load the Simulation  
(you can also try to regenerate the Simulation Engine, only if you have  
Visual Studio installed)



Wait until the viewer application (STK Active-X) pops-up and loads the database.  
Once the 3D view is displayed, you can start the simulation into vsTASKER.

When you select an entity into vsTASKER map, the focus is set on the STK viewer.  
You can see some logics into the UAV and the Launcher as these two entities are  
powered by vsTASKER.

## **Delta 3D Sample**

# **Delta 3D Sample**

For this demo, you need **Delta3D 2-4-0**.

Make sure that **DELTA\_ROOT** environment variable is set to the root directory and that the path can access the Delta3D dll.

Add that to the user path: **%DELTA\_ROOT%\build\_vs2008\bin;%DELTA\_ROOT%\ext\bin**

Do not mix OSG 2.8 and Delta3D dll, change OSG directory name before running any Delta3D sample.



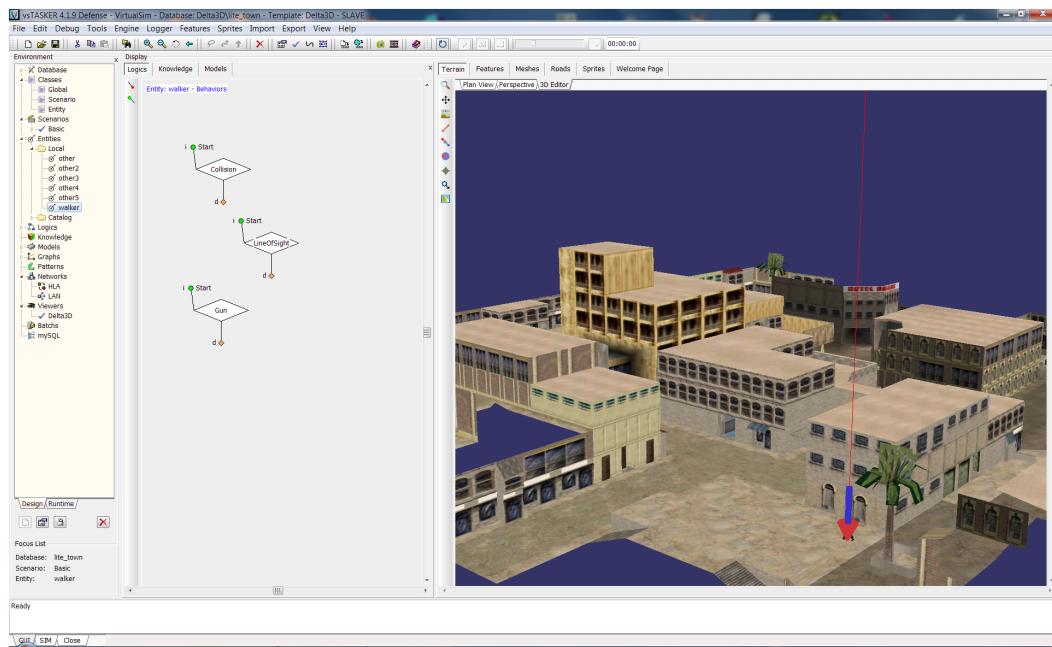
# Little Town

Open [little\\_town](#) scenario.

You can select then tab 3D Editor to switch to the OSG view.  
Use the mouse to change the view and the mouse roll to change the zoom.

Load (or compile) the Simulation Engine.  
You will need a joystick or game pad to run the demo.

Press **O** to get the Head View.  
Have a look at [ScnCam](#) logic for all keyboard shortcuts.  
Use the joystick/gamepad to move around.  
To shoot, depress the joystick **Fire** knob.



## Di-Guy Sample

# Di-Guy Sample

These demos need **Di-Guy** from Boston Dynamics and **VegaPrime** from Presagis.

It shows how to integrate Di-Guy libraries and access the Di-Guy API and internal data during the simulation, from logics and vsTASKER user code.



# First Shooter

Open [bdi\\_stage7](#) in [data/samples/integrations](#) database.

Runaway animates 100 Di-Guy entities into a scenario involving a helicopter landing and crashing.

[Bdi\\_stage7](#) is a First Person Shooter that is controlled from a joystick or gamepad (Logitech USB). The purpose is to find enemies and shoot at them without being killed.

You need to press button 3 to aim (and button 3 again to walk). Use the trigger button to fire.

For the integration, base Entity Class got the following defined:

```
vpDiguyCharacter* diguy_man;
diguyCharacter* man;
```

These pointers are initialized in vsTASKER the same way as it is from a Di-Guy user module:

```
diguy_man = new vpDiguyCharacter();
diguy_man->setName(getName());
diguy_man->setCharacterType("afghan_child_crowd");
diguy_man->setAppearance("afghan_girl2");
```

In most of the Logics, controlling Di-Guy character is done using direct API:

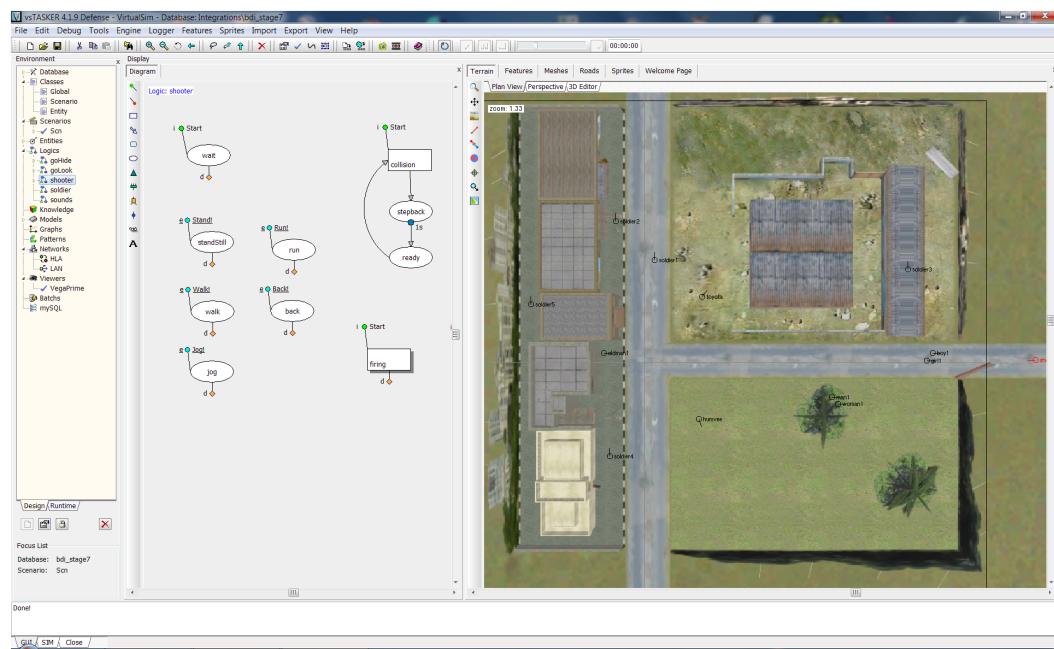
```
E:man->force_action("stand_ready", DIGUY_DEFAULT_FLOAT, 1, 0.0f);
```

In the Database Settings, Compilation and Link pane, specific header files and libraries are used to compile and link the vsTASKER code including Di-Gui API and VegaPrime API to generate the final executable.

These demos can be recompiled from vsTASKER or/and debugged from [vst\\_vega.sln](#) found in [/Runtime/DiGuy](#) directory.

Because VegaPrime is used with Lynx, [.ACF](#) files are needed ([/Runtime/DiGuy](#)). These ACF files are using some models (FLT) and sound files (WAV) that might not be distributed with vsTASKER for copyright reasons.

## First Shooter



# GL-Studio Sample

For this demo, you need **GL-Studio** v4 with a valid license.



# Altimeter

Open [HMI/gls\\_altimeter](#) database.

In Models, you will find the two GL-Studio components loaded ([\\_adi](#) & [\\_altimeter](#)). These two components can be found in [runtime/glstudio](#). You can open the [.gls](#) file to modify the content.

Each vsTASKER GL-Studio component encapsulates the graphical object itself. The component is able to parse the gls file to extract variables (properties) and callbacks.

Using graphical properties is straightforward as shown in the both components (see pane [Code::Runtime](#) of both [\\_adi](#) & [\\_altimeter](#) components).

In order for the graphical object to send callback event to vsTASKER, the following code must be added inside all GL-Studio object callback:

```
vt_rtc->raiseEvent(self->InstanceName(), T_Ptr, self);
```

and in file ....cpp, add the following:

```
#include <vt_rtc.h>
```

You can then compile the demo (force regenerate check button in each GL-Studio form force GL-Studio to regenerate the code of the object before vsTASKER compile it. It is not necessary to do so if the [gls](#) file is left untouched).

Load the simulation engine and start it.

Two little window will appear on the top left corner. [Adi](#) and [Altimeter](#) windows.

If you select player ([Scenario::Scn::Player](#) in Environment), you will see that [Lgk](#) logic behavior symbol is magenta (running).

Open it and start playing with knob and buttons of the Altimeter to see the effect on the two Actions.

# RotorLib Sample

This demo needs **RotorLib** 1.3 (from RT::Dynamics Inc).



The purpose of this integration is to show how to embed a third party helicopter high fidelity model into vsTASKER and control it using logic.

- 1) Open the database `rotorlib` in `data/samples/integrations` directory.
- 2) Compile the generated code then produce the Simulation Engine.  
Make sure that you have installed **RotoLib\_SDK** and that the environment variable `%ROTORLIB_HOME%` is properly set.

- 3) Start the simulation.

Select scenario `Scn` and select the Helicopter on the map.

Choose then Logics pane and double-click on `Canyoning` behavior. You will be able to monitor the execution of the logic in runtime.

The simple logic finds a canyon and makes the helicopter dive into and follow it continuously.

# VR-FORCES Sample

These demos need **VR-FORCES** 3.10 (minimum) from Mäk Technologies.



## • **vrf\_tasking**

This demo shows a tight integration of vsTASKER & VR-FORCES by allowing vsTASKER to behave like a doctrine editor on top of VR-FORCES that is then used like a simple CGF & GIS display.

Open Global window in [Environment::Classes::Global](#).

Select [Definitions](#) pane. Here, you will see all the VR-FORCES header files needed in vsTASKER to access the VR-FORCES API.

In [Declarations](#) pane, you will see a function prototype to let vsTASKER being informed of any entity added in VR-FORCES scenario (manually or from start). [Methods](#) Pane develops the function.

Open Scenario window in [Environment::Classes::Scenario](#)

Select [Declarations](#) pane. Here, you see the main VR-FORCES main node (object): `CtCgf`. This object (pointer) will be used in vsTASKER to access the VR-FORCES environment as it would be from a normal C++ user-module (doing class extension). The three `DtVrfObject` are just there for the purpose of the demo. Not mandatory.

In [Initialization](#) pane, you will see in the case INIT part that `cgf` pointer is set from `udata(). udata()` is attached data to the INIT event sent by a C++ user-module that makes the link between VR-FORCES world and vsTASKER world (in terms of start-up and runtime master-slave issues). This user-module can be found in [Runtime/VrForces/vrfapp.cxx](#).

This module is compiled with vsTASKER generated files so, user can start adding some specific code in it (as long as this code is generic enough. Specific code must be put into vsTASKER, at Global, Scenario or Entity level).

`Cgf->vrfObjectManager()` registers the vsTASKER add entity callback function. Anytime VR-FORCES will add a new entity, vsTASKER will be notified.

Open Entity window in [Environment::Classes::Entity](#)

In the Declarations pane, you find a `DtVrfObject` pointer to the VR-FORCES object. Every entity has their pointer to their VR-FORCES counterpart. Entity class also holds several task objects to be used in Logics.

Open now Scenario property window (double-click background of `Scn` scenario map) and select [Reaction::Events](#) pane. See how `addEnt` event is processed. Here,

according to received DIS data, the proper Entity Catalog is used to create scenario entity.

Expand [Entities::Catalog](#). `vrfTank` and `vrfTruck` have different behavior (select them and Logics). These behaviors trigger different logics defined in Logics.

Open now, for i.e. the logic “circulate” and see how `gotoBeta` Action has been defined (Implementation pane) based VR-FORCES use of predefined Tasks.

If `MAK_VRFDIR` and `MAK_VRLDIR` environment variables are correctly set, you can compile the scenario (tested with **vrforces** 3.10 & **vrlink** 3.10)

Start the simulation. You can monitor for each selected entity, their behavior (click on the entity and select Logics section on top of the Map). The Behavior in magenta is the running one. You can stop it and activate another one. You can also stop the simulation, change the connection, save and start again (no need to recompile).

## • **vrf\_mission**

You need to unzip in VR-FORCES scenarios directory the [vsTaskerScn](#) one provided in [Runtime/VrForces/vsTaskerScn.zip](#). You have there the three scenarios that can be loaded from VR-FORCES (assuming the Mäk-Land terrain is available).

Anytime you do a change (move/create/delete entities on VR-FORCES), you can update the vsTASKER scenario using Menu options: [Import::Scenarios::VR-FORCES](#).

In the three examples provided here, vsTASKER is holding the Logics and sequences basic tasks using the VR-FORCES API and library.

If you open any of the Logics, you will find intensive use of Entity methods: `ent()` - `>methods()` and these methods have been defined in [Sources::Entity](#). These methods contain direct calls to VR-Forces API (see Definitions and Methods panes).

- 1) Open the database [vrf\\_mission](#)
- 2) Compile the generated code.
- 3) Launch the generated VR-FORCES Simulation Engine.
- 4) Start VR-FORCES GUI
- 5) Load the requested scenario (helicopter is the recommended)
- 6) Load the Stealth (if any)
- 7) Start VR-FORCES exercise and monitor the Stealth, the VR-FORCES scenario Map and the vsTASKER logic.

## STAGE Sample

# STAGE Sample

PRESAGIS

This demo needs **STAGE 4** (minimum) from Presagis.

The setup is a little more complicated as the code must be linked with the STAGE libraries.

If you do not have STAGE and a valid license, you cannot run this demo.

**1)** The needed environment variables are set in file `engine\stage\setup.cmd`. Open it and change the STAGE directory setting. Then, you can either copy all these environment variables to the systems one or work from a console window.

**2)** Open a console window. Type: `cd D:\VirtualSim\vsTasker\7`

Then type: `runtime\stage\setup.cmd`

Then type: `start /b vtasker`

Then after that: `start /b stage.de` (you need to have a valid license for STAGE)

**3)** Build the two STAGE scenarios as follow:

- Start STAGE DE.
- Create 2 sensors (radar and visuel)

Radar: Min azimuth: -80, max azimuth: +80, detection: 100% from 0 to 5 km

Visuel: Min azimuth: -80, max azimuth +80, detection: 100% from 0 to 1 km

- Create 1 weapon (missile) type Tracking, with the following dynamic:

Max speed: 200, climb/dive rate: 500 m/s.

- Create 4 platforms (F16, MIG, site and tank)
- Give to F16 and MIG a sensor named radar (profile radar) and 10 weapons named missile (profile missile).
- Give to tank a sensor named visuel (profil visuel) and 2 weapons.

### • Scn 1

Create a scenario Scn\_1

Set the MAP size to 10 km

Drop a platform tank in the scenario and a platform site at 1.5 km north-east of the tank.

### • Scn 2

Create a scenario Scn\_2

Set the MAP size to 20 km

Drop a platform F16 in the scenario, positioned south and facing north.

Drop a platform MIG in the scenario, positioned north-east and facing west, color RED.

Set the two platforms initial speed: 100 m/s and elevation 2000 m

**4)** In the vsTASKER Database Properties window, check in Compilation and Link tab that the path is correct for the Visual C++ compiler and linker.

**5)** Generate and compile the vsTASKER stage database.

If you have errors, try to correct them. Probably that STAGE environment variables are not correctly set. This demo works with the STAGE out of the box, not the extended one.

**6)** Start STAGE DE and load the **tasker.dbs** database.

**7)** Launch the simulation engine (vsTASKER is in slave mode for this demo and thus, STAGE DE must start the simulation).

**8)** Select [Scn\\_1](#) in STAGE and start STAGE. If you want to monitor the logic in vsTASKER, select the same scenario and wait for the entity to turn magenta before clicking it, then the logic to monitor the flow.

You can then stop the simulation (still using STAGE) and try [Scn\\_2](#) that shows a little dog-fighting between two entities.

## Flsim Sample

# Flsim Sample

PRESAGIS

This demo needs **FLSIM 9.0** from Presagis. The setup is a little more complicated as the code must be linked with the FLSIM libraries.

If you do not have FLSIM and a valid license, you cannot run this demo.

1) The needed environment variables are set in file `engine\flsim\setup.cmd`. Open it and change the FLSIM directory setting. Then, you can either copy all these environment variables to the systems one or work from a console window.

2) Open a console window. Type: `cd D:\VirtualSim\4.1\vstasker`

Then type: `runtime\flsim\setup.cmd`

Then type: `start /b vstasker`

Then after that: `start /b flsimde` (you need to have a valid license for FLSIM)

3) Go to vsTASKER and open the database `flsim.db` (in `D:\VirtualSim\vsTasker\7\data\db\integrations`).

## • Landing Scenario

1) In the tree-list, select scenario `Test` and compile the scenario. In the build output window, the last line should be: `link.exe @c:\temp\nma01876`.

If not, try to understand errors. Probably dues to bad FLSIM installation or an error in the FLSIM environment variables.

2) Launch the FLSIM\_SIM generated executable (vsTASKER is in slave mode for this demo and thus, FLSIM DE must start the simulation). Ignore the popup window (close it).

3) Go to FlsimDE (you have started it previously from the command shell). In menu `Aircraft`, select `Identification`. Then from the list, load the `F16`. Now, add a system. Select any system in the list (same window) then click button `Add After`. In the popup window, select `vsTASKER_Model` and click `Ok` then `Close`.

4) In menu `Configure`, select `Initial Condition`. In the popup window, click `In Air` (in `Position` pane)

5) In menu `Control`, select `Execution` then click button `Run`. Set the focus in the `Flight_Scene` window and press space bar (do not touch the mouse for 1 second).

6) You should be flying north at 2000m and the aircraft should then turn right.

## Flsim Sample

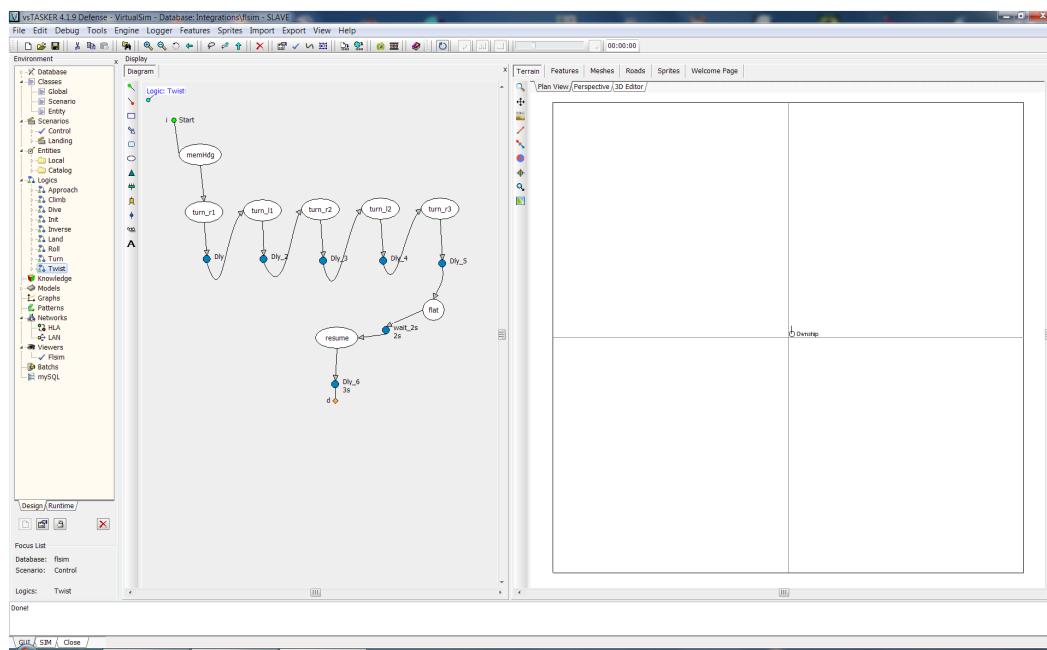
- 7) Go back to vsTASKER and select scenario [Test](#). Then double-click the magenta aircraft symbol then double-click the magenta triangle (approach) running logic. You are able to monitor the do-list. When the [Approach Logic](#) is performed, select aircraft (in the Entities tree-list) then double-click logic [Land](#) and observe the logic flow.
- 8) To restart this simulation. Just focus on the [Flsim\\_Scene](#), press ESC (or press button Stop in the FlsimDE execution window and press [Start](#) again).

## • Control Scenario

Now, restart this simulation (steps 1 to 7) with the scenario [Control](#). You will use the pop-up window.

When the aircraft will be flying in FLSIM, click on any of the pop-up window button to initiate a maneuver.

When the maneuver is over (or seems to be), click the reset button than select another one.



# Helisim Sample

This demo needs **HLSIM 5.0** from Presagis. The setup is a little more complicated as the code must be linked with the HLSIM libraries.



If you do not have HLSIM and a valid license, you cannot run this demo.

**1)** The needed environment variables are set in file `engine\hlsim\setup.cmd`. Open it and change the HLSIM directory setting. Then, you can either copy all these environment variables to the systems one or work from a console window.

**2)** Open a console window. Type: `cd D:\VirtualSim\4.1\vstasker`

Then type: `runtime\hlsim\setup.cmd`

Then type: `start /b vstasker`

Then after that: `start /b hlsimde` (you need to have a valid license for HLSIM)

**3)** Go to vsTASKER and open the database `hlsim.db` (in `D:\VirtualSim\vsTasker\7\data\db\integrations`).

**4)** In the tree-list, select scenario `Test` and compile the scenario. In the build output window, the last line should be: `link.exe @c:\temp\nma01876`.

If not, try to understand errors. Probably dues to bad HLSIM installation or an error in the HLSIM environment variables.

**5)** Launch the HLSIM\_SIM generated executable. Two consoles should appear and one pops up a Malfunction Window (vsTASKER is in slave mode for this demo and thus, HLSIM DE must start the simulation).

**6)** Go to HlsimDE (you have started it previously from the command shell). In menu `Helicopter`, add a system. Select any system in the list (same window) then click button `Add After`. In the popup window, select `vsTASKER_Model` and click Ok then Close.

**7)** In menu `Control`, select `Execution` then click button `Run`. Set the focus in the `Flight_Scene` window and press space bar (do not touch the mouse for 1 second).

**8)** The helicopter should take-off and stabilize after hovering shortly.

**9)** Go back to vsTASKER and select scenario `Test`. Then double-click the magenta helicopter symbol then double-click the magenta triangle (take-off) running logic. You are able to monitor the do-list.

**10)** Press the button `Send Malfunction` in the Malfunction window. You should see the helicopter spinning crazily and finally crash.

## Helisim Sample

**11)** To restart this simulation. Just focus on the [Flsim\\_Scene](#), press ESC (or press button Stop in the FlsimDE execution window and press Start again).

## Trinigy Sample

# Trinigy Sample

This demo needs **VisionSDK** from **Trinigy**.



It shows how to integrate and control animated characters from the VisionSDK library and display them using Trinigy graphic engine.

Open [warriors](#) database in [data/samples/integrations](#) directory.

[Simple](#) scenario shows several animated characters (Elf Warriors) provided by Trinigy that obey to simple rules coded into vsTASKER.

For the integration, [BaseEnt](#) and [AnimatedEnt](#) Classes have been defined as child of base Entity Class. In each of them, specific data and methods have been defined. In Global code, the [TrinigyAppInit](#) function is called from the main loop defined in: [vst\\_trinigy.cpp](#) found in [/Runtime/Trinigy](#).

For the integration, vsTASKER holds pointers to data instantiated using VisionSDK library:

```
pCharacter = (AnimatedCharacter_cl *)
    Vision::Game.CreateEntity("AnimatedCharacter_cl",
                                VisVector_cl(0,0,0),
                                "models\\ElfWarrior.model");
pEntity = (Vis BaseEntity_cl*) pCharacter;
```

Logics and Knowledge are accessing directly the API of VisionSDK to control the animated entities:

```
void Ent::attack() {
    pCharacter->GetUpperBodyState()->Attack();
}
```

In the Database Settings, Compilation and Link pane, specific header files and libraries are used to compile and link the vsTASKER code including VisionSDK libraries to generate the final executable.

This demo can be recompiled from vsTASKER or/and debugged from [vst\\_trinigy.sln](#) found in [/Runtime/Trinigy](#) directory.

# **My First Simulations**

Creating a simulation with vsTASKER is **easy, straightforward** and **powerful**.

Nevertheless, basics must be acquired through practice (and training) to create ground for quick and efficient development.

In the following step by step samples, we will go from the simplest simulations to the more complex ones.

This will give you the flavor of what can be done with this product.

You can see some introduction videos here: [virtualsim.com/category/tutorials](http://virtualsim.com/category/tutorials)

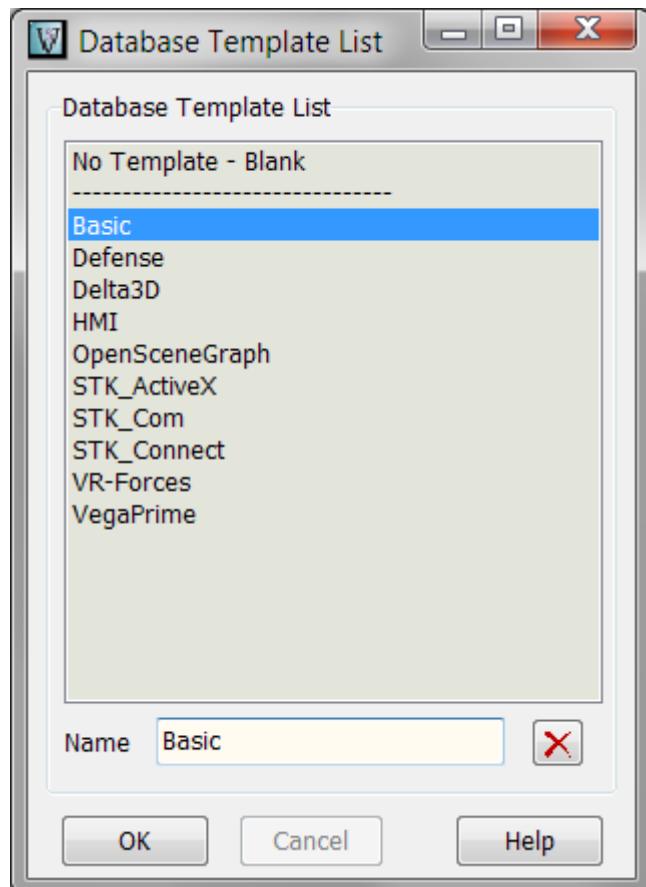
## Simple Entity

# Simple Entity

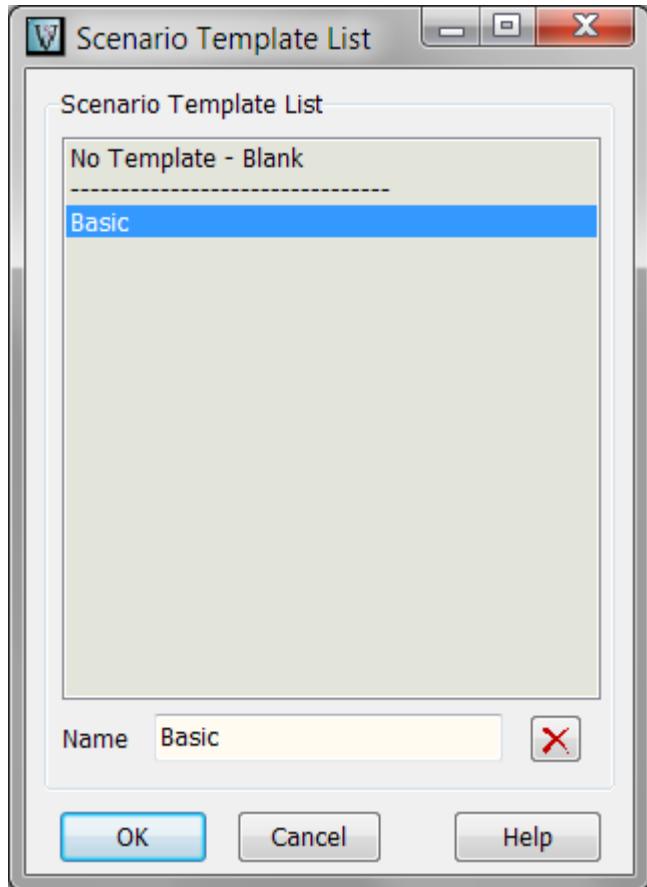
Let's create a basic Entity that will move in one direction.

First, let's create a [simple\\_entity](#) database.

Click  then answer Yes at [Create a New Database](#)".



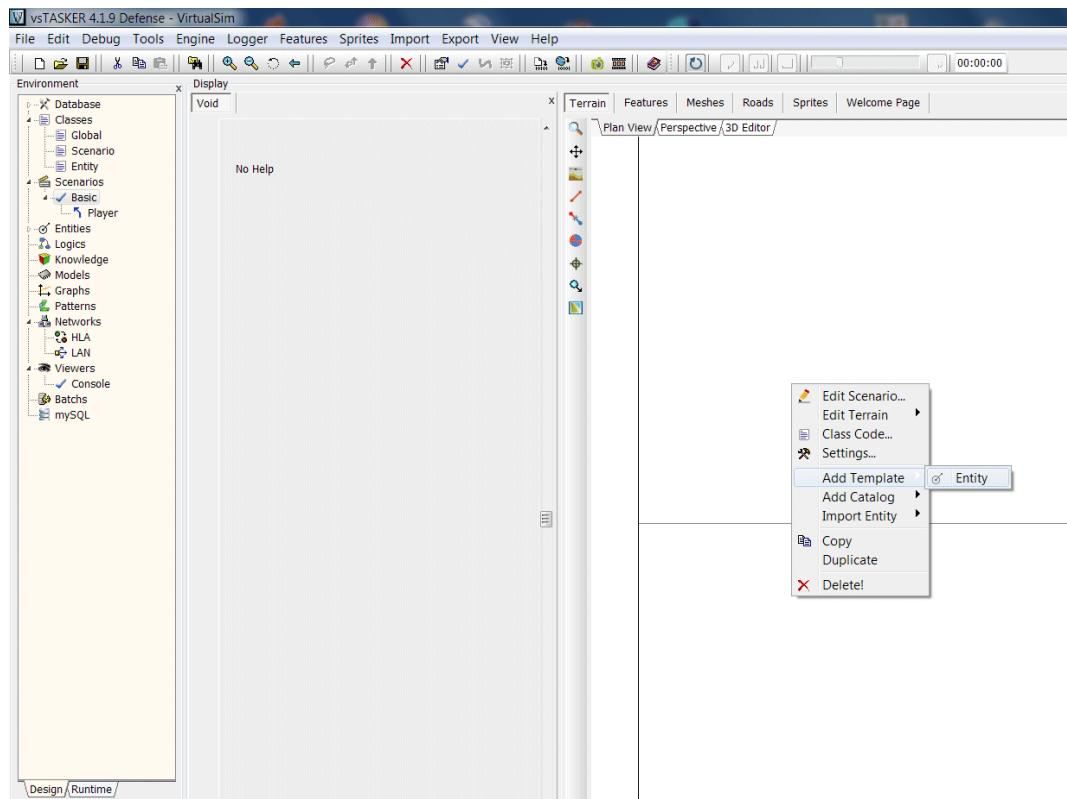
The Database Template selector displays all available built-in Templates that facilitate database creation. For more on Templates, click [here](#).  
Select [Basic](#) then OK (or double-click Basic).



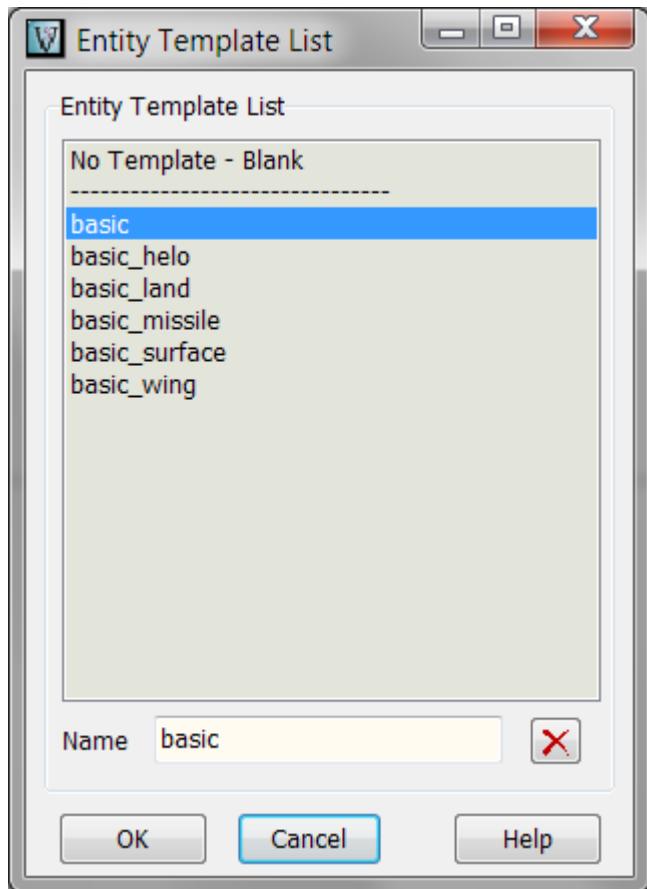
The Scenario Template selector lists all Scenario templates recorded inside the **Basic** Template.

Select **Basic** then OK (or double-click **Basic**)

## Simple Entity

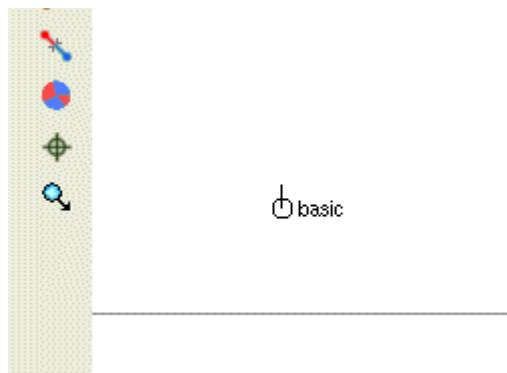


To add an Entity on the terrain, right-click over the map (at the position you want to add the Entity) then select [Add Template -> Entity](#).



The Entity Template selector lists all Entity templates recorded inside the Basic Template Scenario.

Select Basic then OK (or double click Basic)



Basic entity appears on the map and can be selected using the mouse (blue circle appears on selected or focused entities).

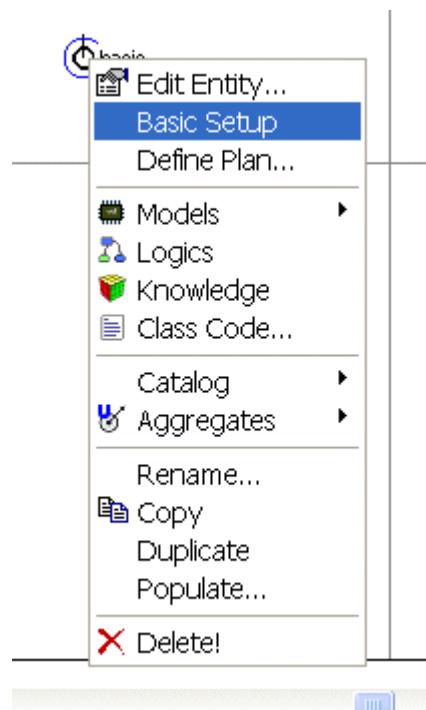
You can also reposition the entity by selecting it and moving over the map by keeping down the right mouse button.

Property Window)

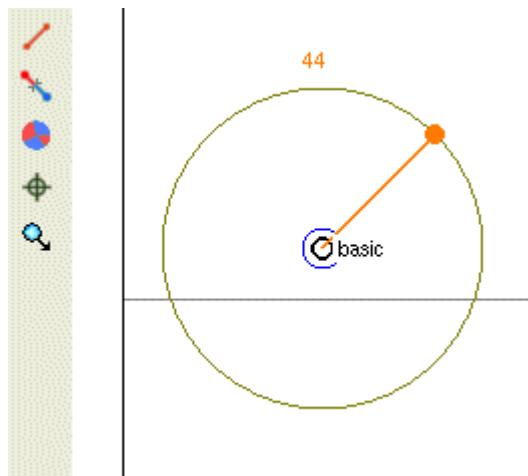
Now, we will set the heading of the entity, visually (this can also be done using the

Select the Entity then right click and select the Basic Setup option:

## Simple Entity



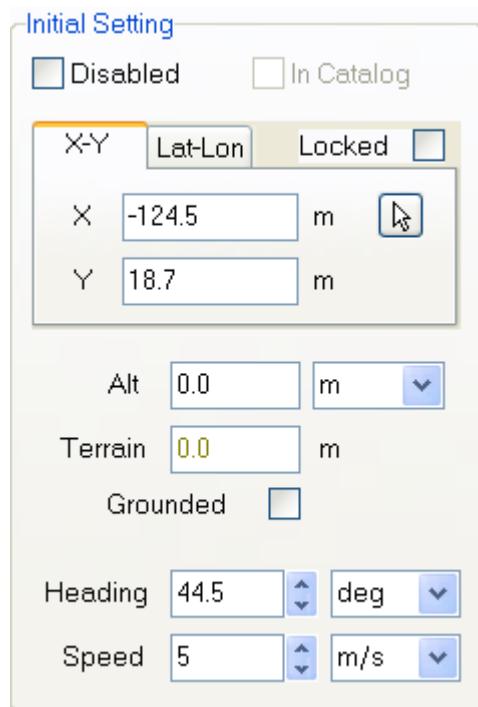
The heading selector is shown in orange. Select the heading handle and, while maintaining down the mouse button, rotate it around the circle at the desired heading.



Now, we need to set the speed of the Entity.

Just double-click the basic symbol displays the Entity Properties window.

## Simple Entity

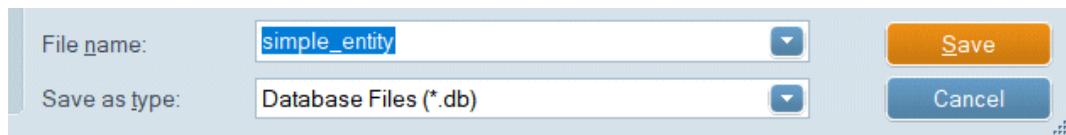


In this window, find Initial Setting pane and set the speed to 5 m/s

Press OK.

Now, let's save the database.

Press and in the file window, write [simple\\_entity](#), then [Save](#):



Now, it is time to generate the code.

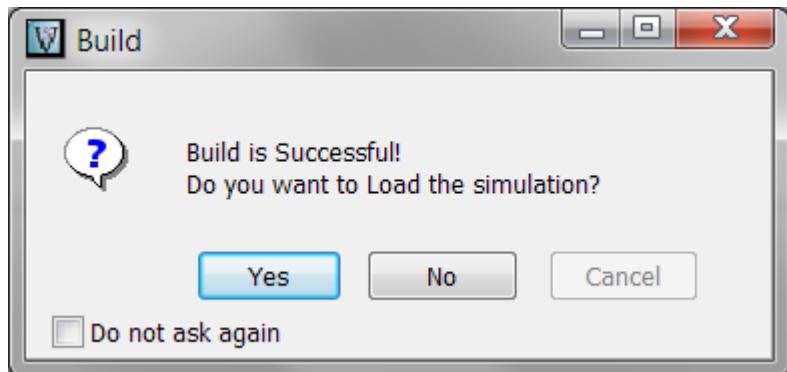
Be sure you have a Microsoft compiler properly installed (Visual Studio Professional or Express).

If you want to know how and where the code will be generated, look at the Database Setup options in the User Manual.

Press .

A Console window will appear and compilation will start.  
You should have no problem compiling at this stage.

## Simple Entity



When build is successful, this means the compilation and link had no error and that the executable is ready to be loaded. Press Yes to load and start it.

```
C:\WINDOWS\system32\cmd.exe
D:\VirtualSim\vsTasker\4.0>D:
D:\VirtualSim\vsTasker\4.0>cd \VirtualSim\vsTasker\4.0\Runtime
D:\VirtualSim\vsTasker\4.0\Runtime>start /b test4.exe
D:\VirtualSim\vsTasker\4.0\Runtime>//start /b vsrad_ppi.exe
Syntaxe du nom de fichier, de répertoire ou de volume incorrecte.

=====
=           vsTASKER 4.0 b31          =
=                                         =
= Copyright (c) VirtualSim Sarl. 2004-2009 =
= All rights reserved.                   =
= Simulation Engine is license free!   =
=                                         =
=      www.virtualsim.com - Nice - France =
=      support@virtualsim.com           =
=====

Having loaded environnement for: test4
Simulation Engine is ready
```

The simulation engine starts as a Console because the Viewer is a Console. For more information on Viewers, click [here](#).

The simulation engine (console application) is connected to the vsTASKER GUI using the shared memory. So, we can start & stop the simulation using the control



To learn more on this toolbar, click [here](#).

Once the simulation has started, you can see the basic entity moving at 5m/s on the given direction.

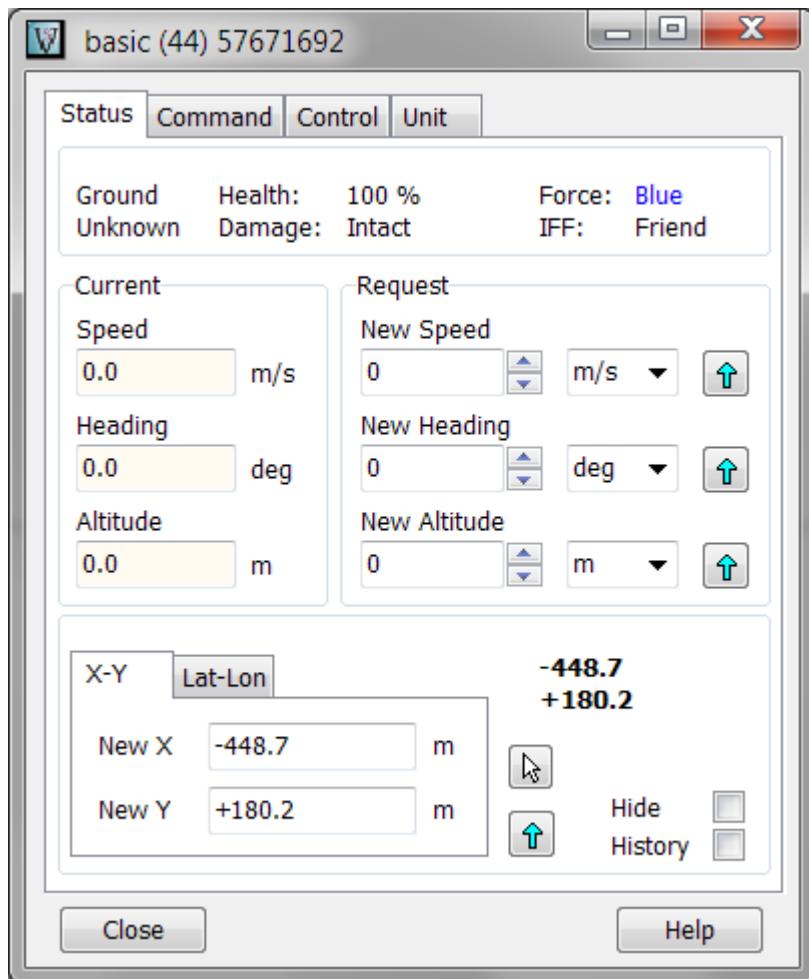
## **Simple Entity**

## Changing Parameters

# Changing Parameters

Now, let's change the speed and heading of the entity during runtime (while simulation is running).

On the map, select the Entity with the mouse and double-click to popup the entity Hook Window:



Now, in the **New Heading** field, enter 180 then send the data using button.  
Do the same with the New Speed by giving 10 or 0 m/s.

To reposition the entity, either use the button , then, click on the map to get the X/Y Lat/Lon position then send it to the Entity using   
(you can also select the entity with the mouse then, while holding down the mouse button, move the entity at the desired position).

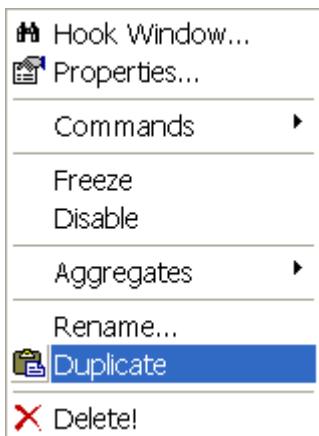
## Changing Parameters

## Cloning an Entity

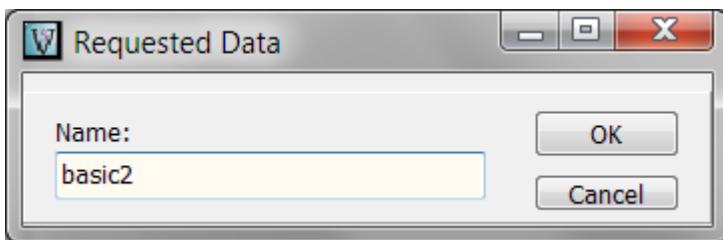
# Cloning an Entity

To finish, let's duplicate the current basic entity.

Select the entity then right-click



and select [Duplicate](#)



New name is automatically given based on the selected entity.

You can change it.

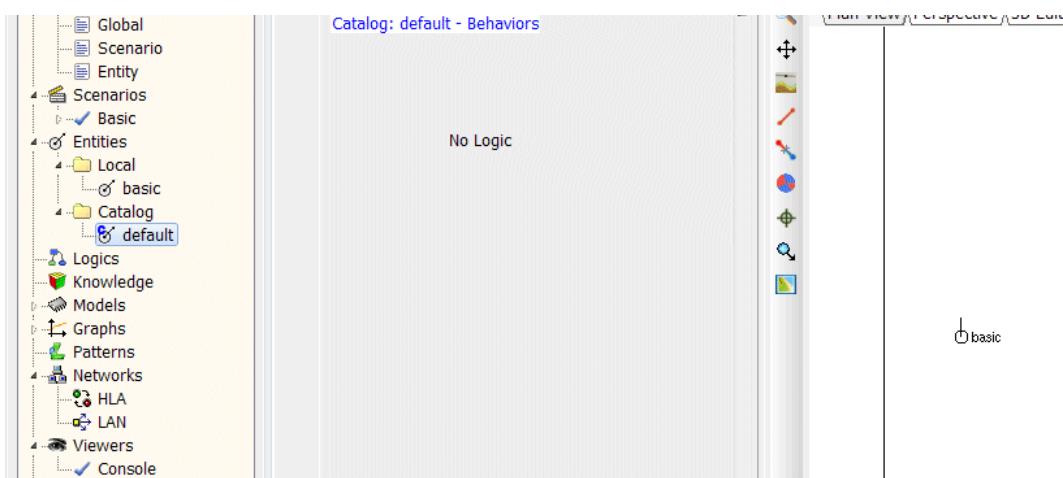
Press OK.

The new [basic2](#) entity will be created on the map as a clone.

Now stop the simulation. Note that runtime entities (created during runtime) are automatically removed.

Populating the Scenario with entities can be done at Design or Runtime by just drag & dropping Entities from the Environment Tree List:

## Cloning an Entity



## Using Plans

# Using Plans

Plans are based on [routines](#) which are listed in the entity class pane, for each entity class type.

A Plan is unique to an entity and can be defined at design or runtime. It is based on predefined routines which belong to the entity class. Thus, each class of entity can have different routines. The inheritance tree allowing basic routines to be shared by all while children classes will either provide specific ones or specialize inherited ones.

A plan must be seen as a string of localized (or not) routines. When the routine is localized, it is attached to a [Plan-Point](#) and the coordinates matters.

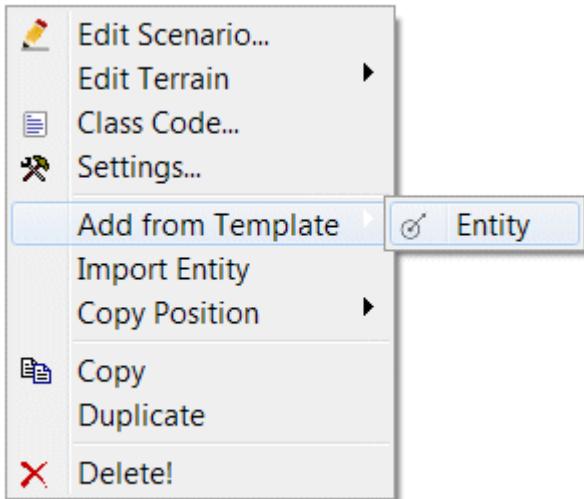
vsTASKER provides some built-in routines that belongs to each Template. They can be imported/exported from/to the [Data/Shared](#) directory.

The simplest routine is the [moveTo](#) that directs the entity towards the plan-point and stops it when reached.

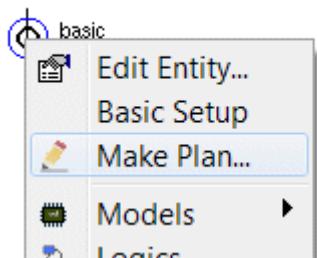
When a routine returns [Done](#), the next plan-point is activated (and the associated routine is executed) until all plan is processed.

# Plan at Design

Here, let's start by creating a simple Plan using default entity:

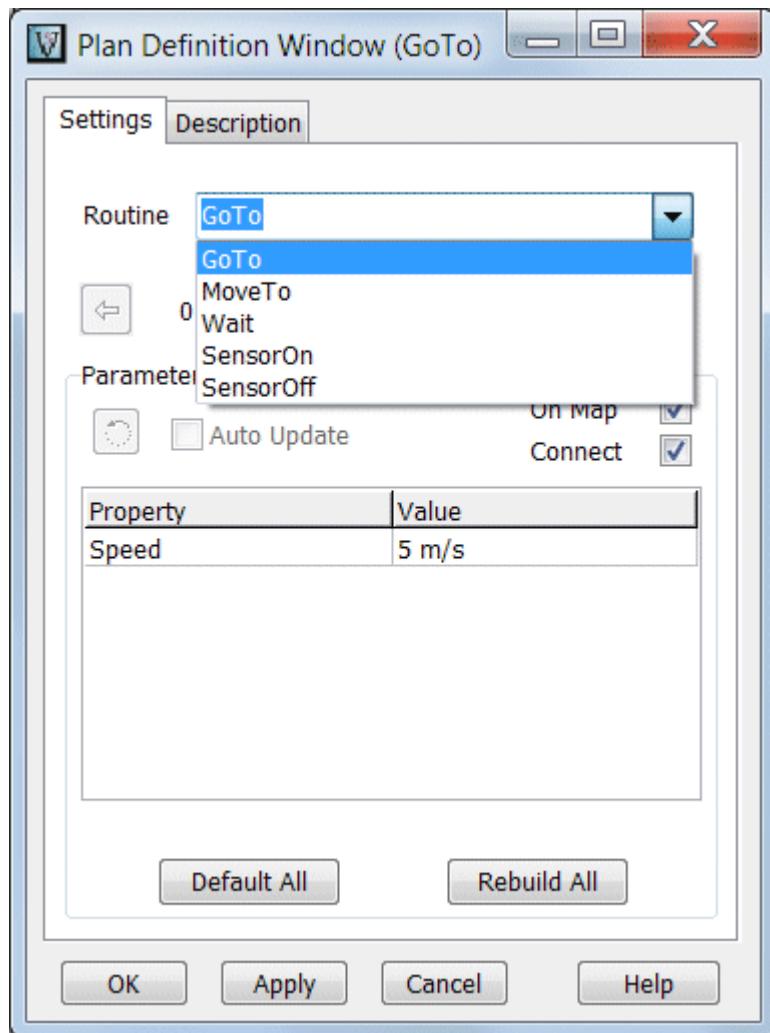


First, let's put one entity on the terrain map



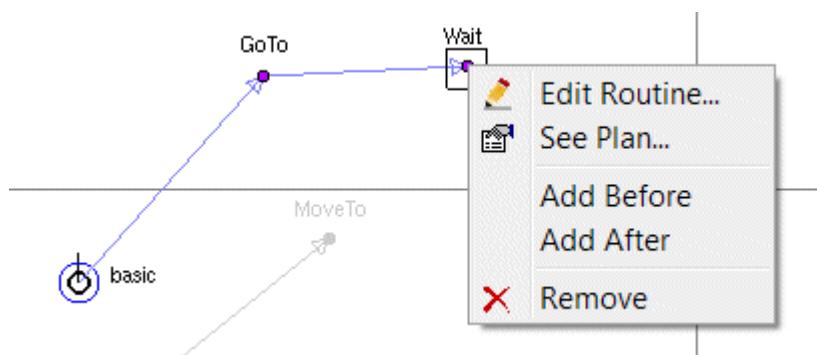
Then, let's select Make Plan to define the plan as a succession of Plan-Points.  
The cursor will change to a cross to invite to click on the map where the plan-point will be put.

## Plan at Design



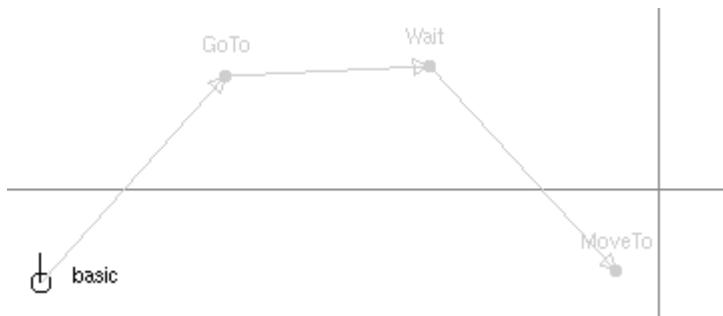
Once the Plan-Point has been localized, the definition window will popup, inviting to select the Routine attached to the Plan-Point.

Let's select **gotoTo** and then let's set the speed to **5 m/s**



We will add another Plan-Point with the **Wait** Routine set at **10 seconds** and we will add a **MoveTo** Plan-Point as below:

## Plan at Design



*As Routines cannot be created/defined at runtime, it is important to make sure they are all available at design time.*

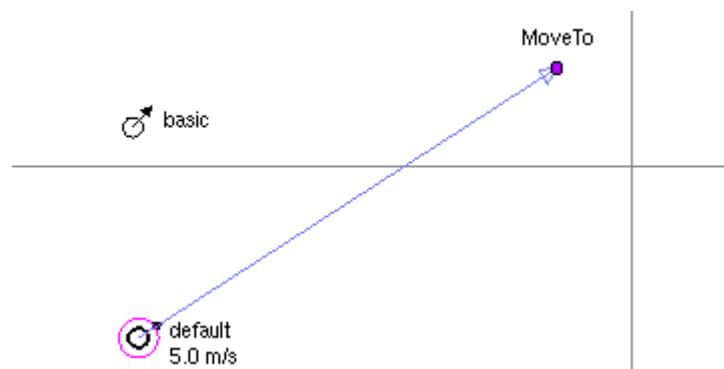
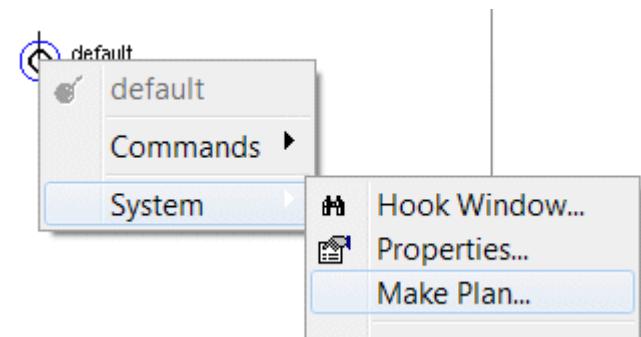
## Runtime Plans

# Runtime Plans

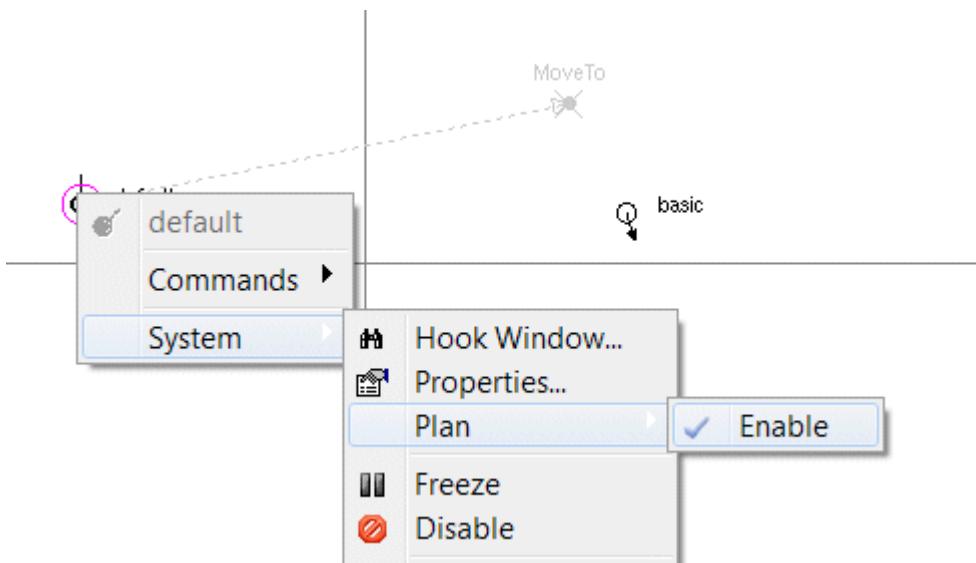
Let's compile and run the simulation.

We can see that while running, the Plan-Point can be repositioned.

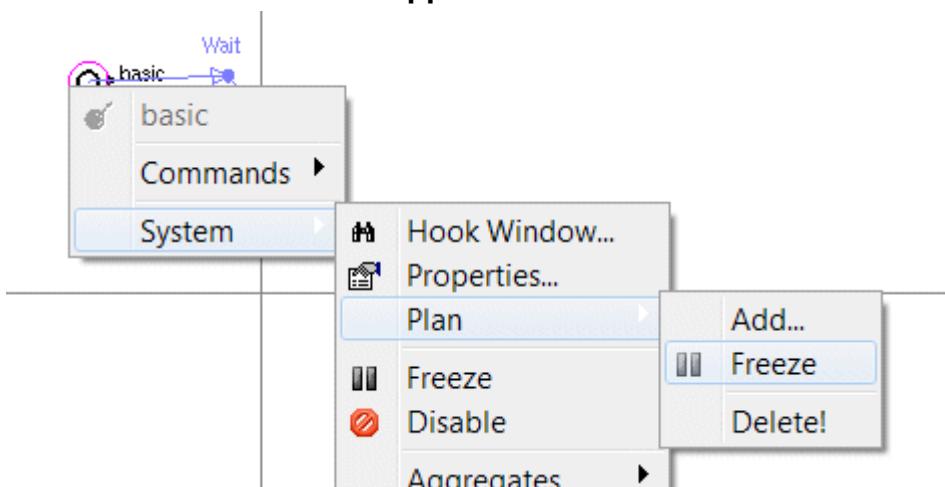
While the basic Entity is following its plan, let's **add** another **Entity** (default) and let's give it a runtime Plan to move somewhere on the map:



If a Plan is **disabled** (at design time), it can be **enabled** during runtime (see below)

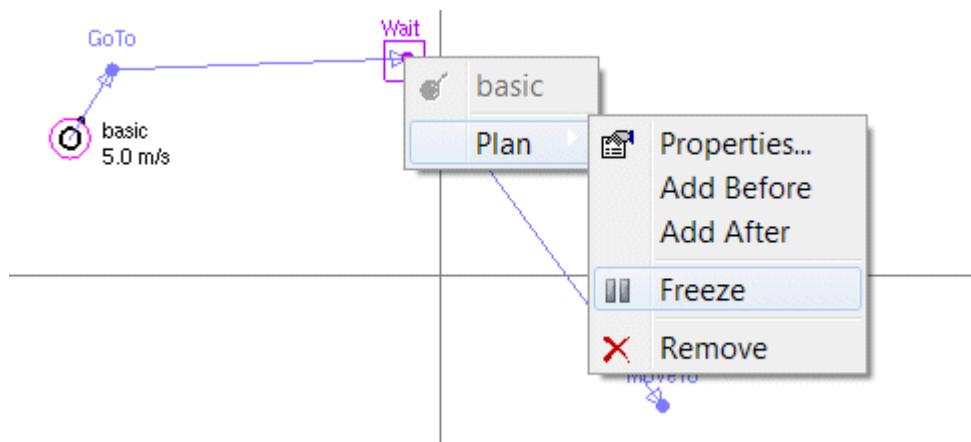


A Plan can also be **frozen/resumed** directly from the popup menu (below). Another Plan-Point can be **appended** or the whole Plan can be **deleted**.



Once a Plan-Point has been selected, another one can be **added before** or **after** it. A Plan-Point can also be frozen/released. When a Plan-Point is frozen, the attached Routine is frozen. That does not mean the Entity will freeze. If the Entity was moving towards the Plan-Point, it will continue and pass-by. If the Plan-Point is resumed, the Routine will start again and might drive the Entity back to the Plan-Point location (if goTo or moveTo).

## Runtime Plans



Entity Plans are very flexible and only require that the associated Routines are already defined and available.



*Plans can also be controlled from Logics (enable, disable, freeze and resume)*

# Simple Logic

For this example, we will create a simple Logic that will make the basic entity of the previous example to move erratically on the map.

First, we need to create a Logic.

Select Logic in the Environment Tree-List then, in the Display diagram section, right click and select Add Logic



The Logic is created.

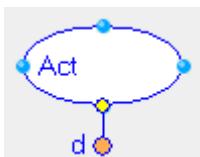
Double-click the Logic symbol (below) to Open it's definition.



Now, you can use the Logic toolbar to create visually the logic.

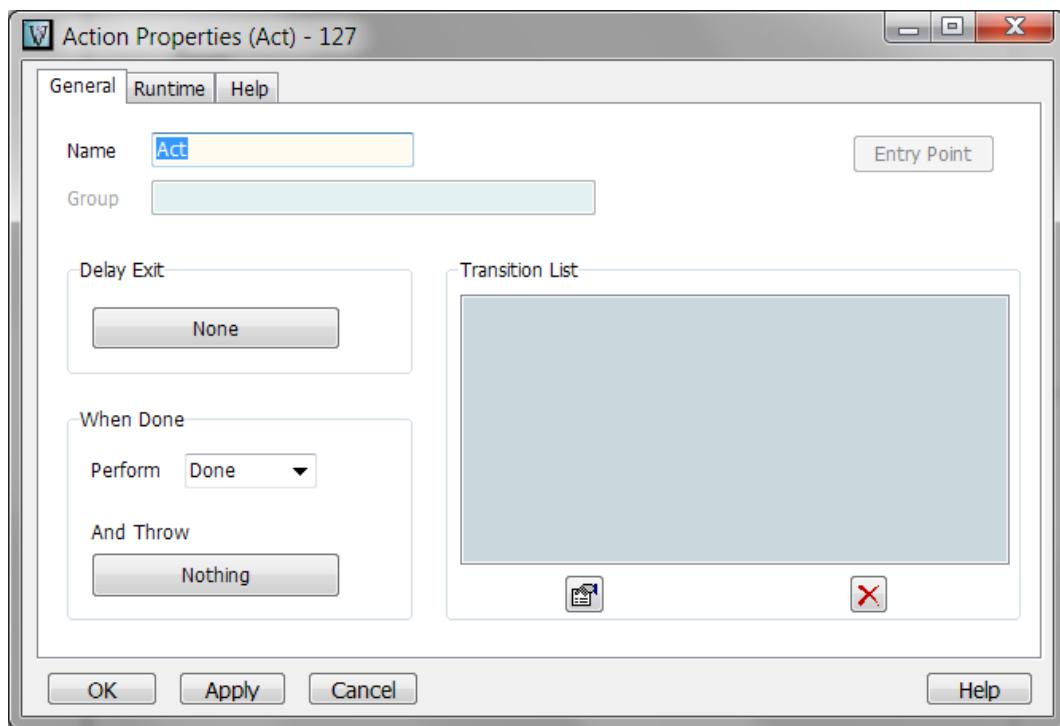
In this case, we will use two objects: Action and Delay.

Click on the Action button  of the Toolbar then drop it on the Diagram



Double click the object to open the Property Window.

## Simple Logic



Set the Name to "divert" then select Runtime pane.

In the Runtime pane, you will enter the code that will be executed (by the simulation engine) as soon as the Action object is triggered.

You can enter here any C/C++ code that will (or not) use vsTASKER SIM API or any other API of third party environment (other CGF, drivers, CGI, etc.)

For this simple example, we want our entity to change its heading and speed, randomly.

Thus, enter the following code:

```
E: dyn->setHeading(DEG2RAD(RANDOM(-180,+180)));
```

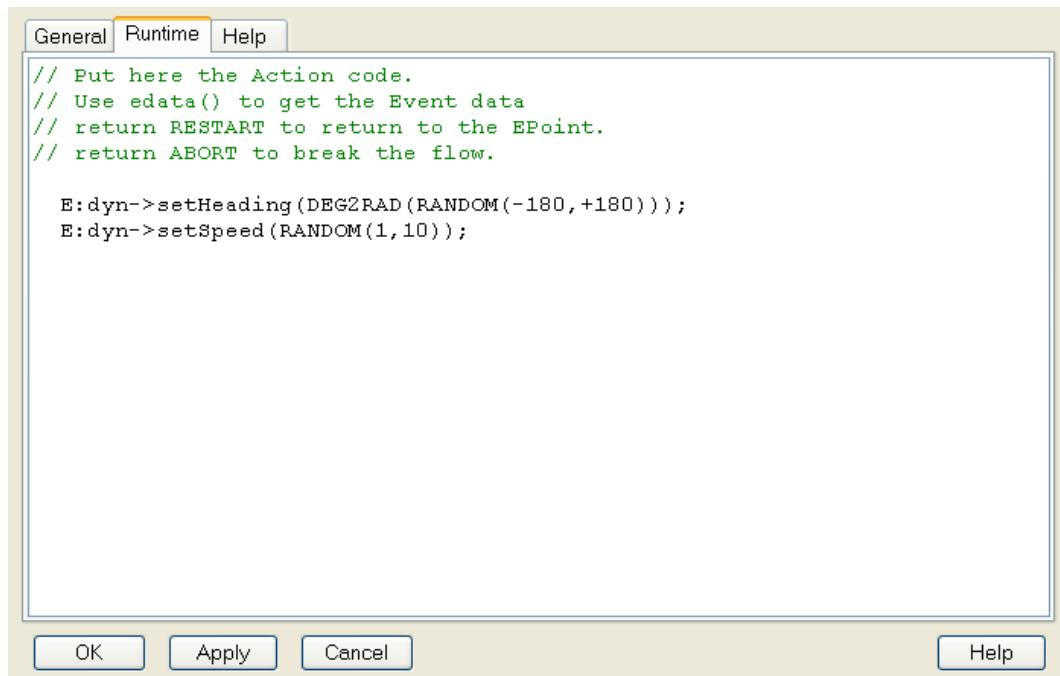
**E:** stands for Entity and is a shortcut to `ent()->`

**dyn:** the dynamic object of the Entity. See Entity Class Declaration pane for more information on this object (defined in Models)

**setHeading()** and **setSpeed()** are virtual functions defined in `model/include/basic_dyn.h`

```
E: dyn->setSpeed(RANDOM(1,10));
```

## Simple Logic

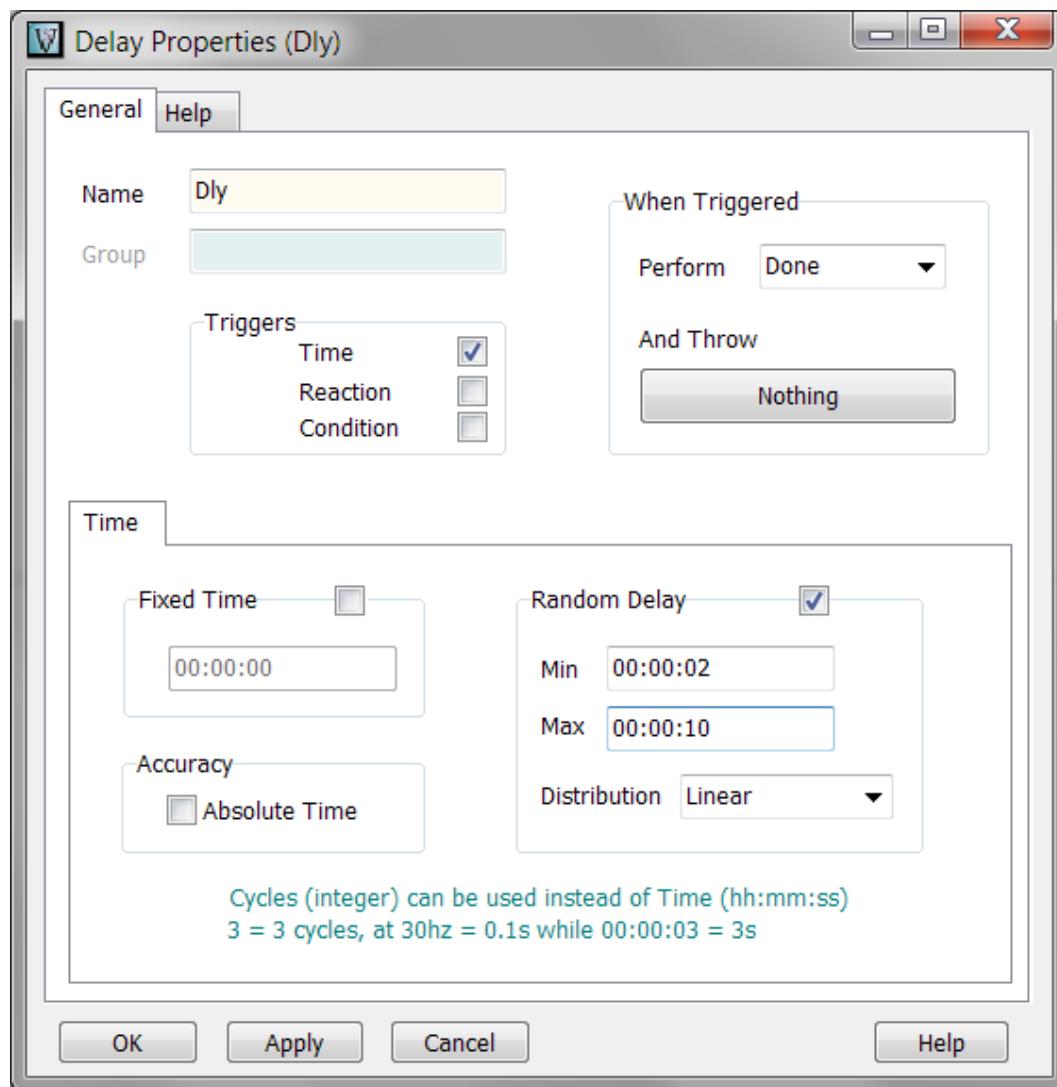


Close (OK) the window then, drop a Delay object below the **divert** Action.



Double click the Delay object to set the random delay time.

## Simple Logic



Here, we want to add a Random Time Delay to stop the logical flux for a random number of seconds before changing again the heading and the speed of the Entity.

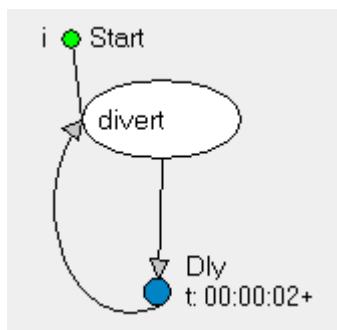
Let's click the Random Delay check button and set the Min time to 3s and Max time to 10s (see picture above).  
Close (OK).

Now, we must add an Entry Point to the Logic. A Logic without Entry Point will never starts. We want to attach the Entry Point to the "divert" Action. To do so, select the Action object then click on the Entry Point  toolbar button.

Then, we must connect the exit arrow of the Action to the Delay object. For that, select the  and drag it close to the Delay object. Release the mouse when over one of the target blue anchor dots.

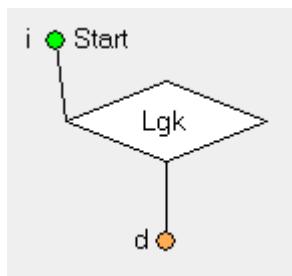
## Simple Logic

Do the same with the Delay exit arrow and return it to the Action object.



Now that the Logic is defined, it must be attached to the Entity.

For that, select the Entity on the Map, then, on the Logic Entity Behavior pane (left of the Map when Entity is selected), right click the mouse, and add the **Lgk** (unnamed) Logic:



Now that the Logic is attached to the Entity, with a Start Entry Point, it will be executed at simulation start.

Time to generate the code, compile, and run the simulation.

During the run, select the Entity, double click the Behavior **Lgk** (see above) and see how the Action and Delay are triggered (use for magenta color for the active object)

## Simple Component

# Simple Component

This sample will show how to create a basic component and attach it to an Entity. The Component will be a 2Hz frequency module that will scan around the attached Entity up to a given distance (user defined) and will show a line to the detected Entity.

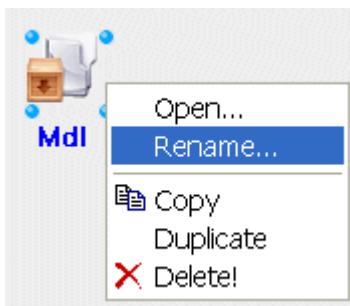
First, we need to create a Model that will hold as many Components (or Objects) as we need. Either we use an existing Model (like Sensors) or we create a new one.

Let's create a new one called: [myModel](#)

Select Models in the Environment Tree-List then, right-click the mouse then, Add a Model, like shown below:



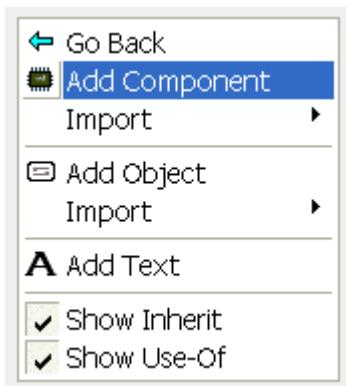
Now, right-click the new Model symbol and select rename.



Rename it to [myModel](#) then, double-click the Model symbol to open it.

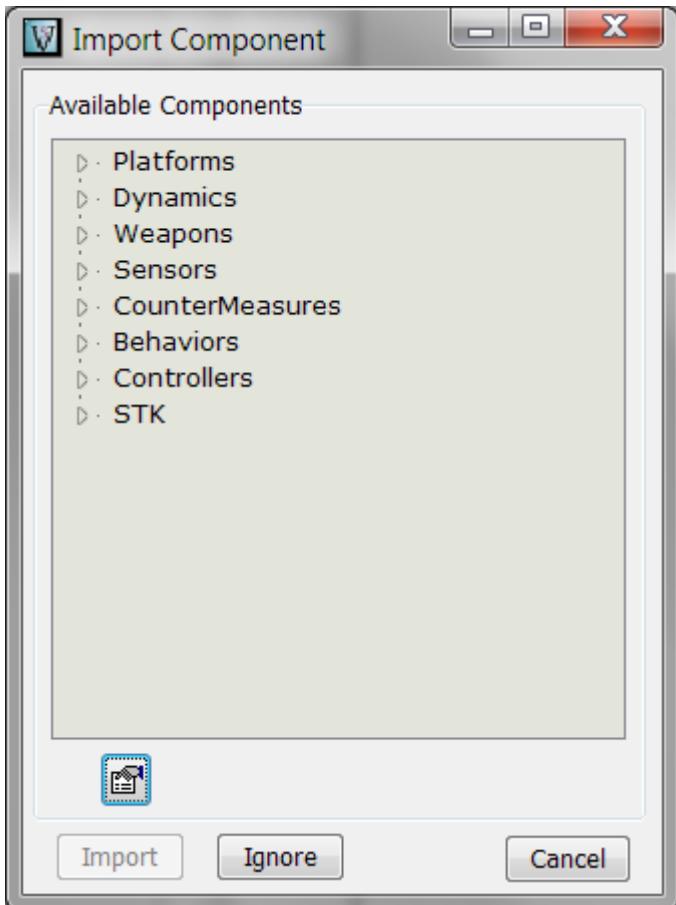
On the diagram, right-click the mouse to add a Component (into the Model)

## Simple Component



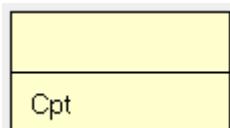
Import Component window allows you to select an existing component from the library.

You do not want that now as you will create a new component from scratch.  
Select [Cancel](#).

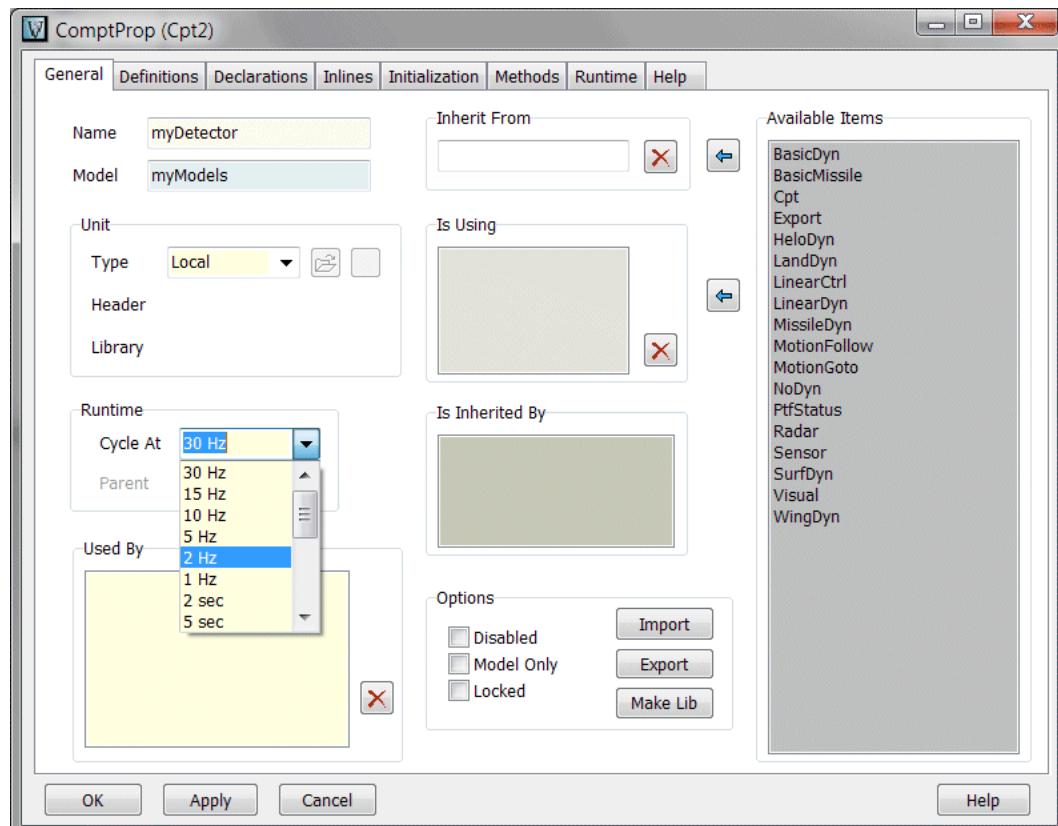


The component is then shown on the Diagram.

## Simple Component



Double-click it to open it.

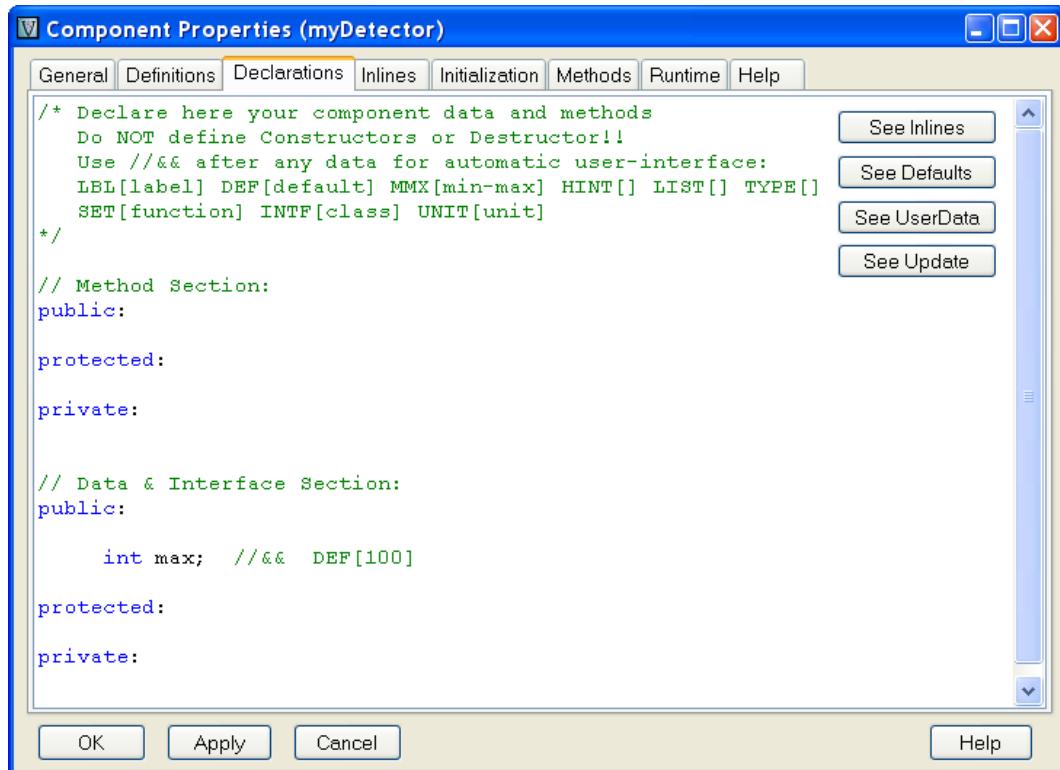


Set the name to **myDetector** and select 2 Hz in the Runtime Cycle at selector, as shown above.

We will define now the algorithm of the component.

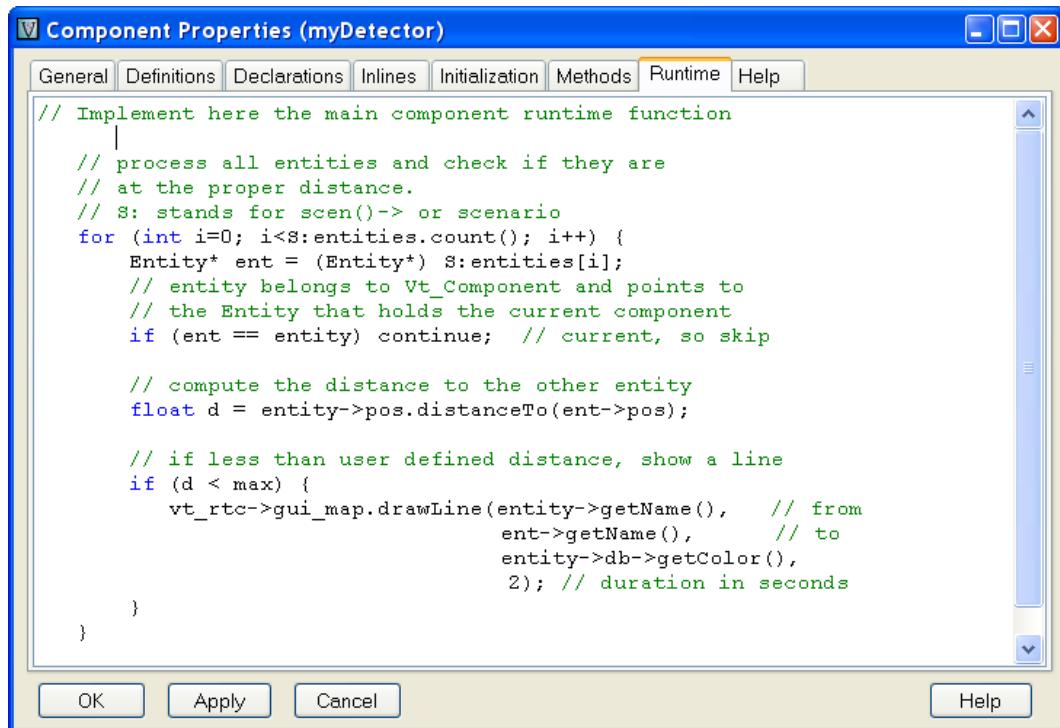
In the Declaration pane, we will add a variable that will define the maximum distance for detection. We will make this variable GUI-dynamic by using **//&&** (see Interfaces)

## Simple Component



max (integer) variable will be added into the Component Interface and defaulted to 100.

Now, let's define the main runtime loop.



## Simple Component

```
for (int i=0; i<S:entities.count(); i++) {  
    Entity* ent = (Entity*) S:entities[i];  
    if (ent == entity) continue;  
    float d = entity->pos.distanceTo(ent->pos);  
    if (d < max) {  
        vt_rtc->gui_map.drawLine(entity->getName(),  
                                  ent->getName(),  
                                  entity->db->getColor(),  
                                  2);  
    }  
}
```

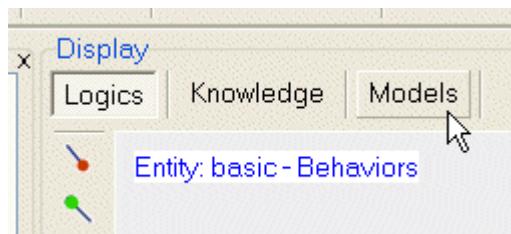
Explanations are embedded into the code. See the Simulation Engine API for a deeper understanding of vsTASKER SIM classes.

Basically, in this algorithm, twice per second, all entities of the scenario will be processed (except the one that holds the current Component) to compute the distance to the component's owner. If the distance is less than the default or user-defined distance, then, a line will be shown on the GUI Map, for 2 seconds.

Now, let's add the component to basic Entity.

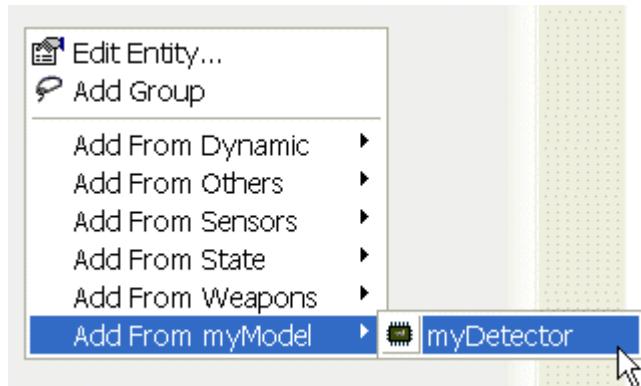
First, select basic Entity into the map.

Then, select the Models Pane on the Display area:

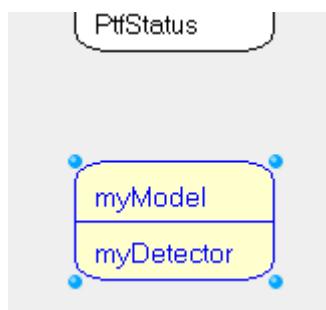


Then right-click the mouse and add the [myDetector](#) Component from the [myModel](#) Model:

## Simple Component

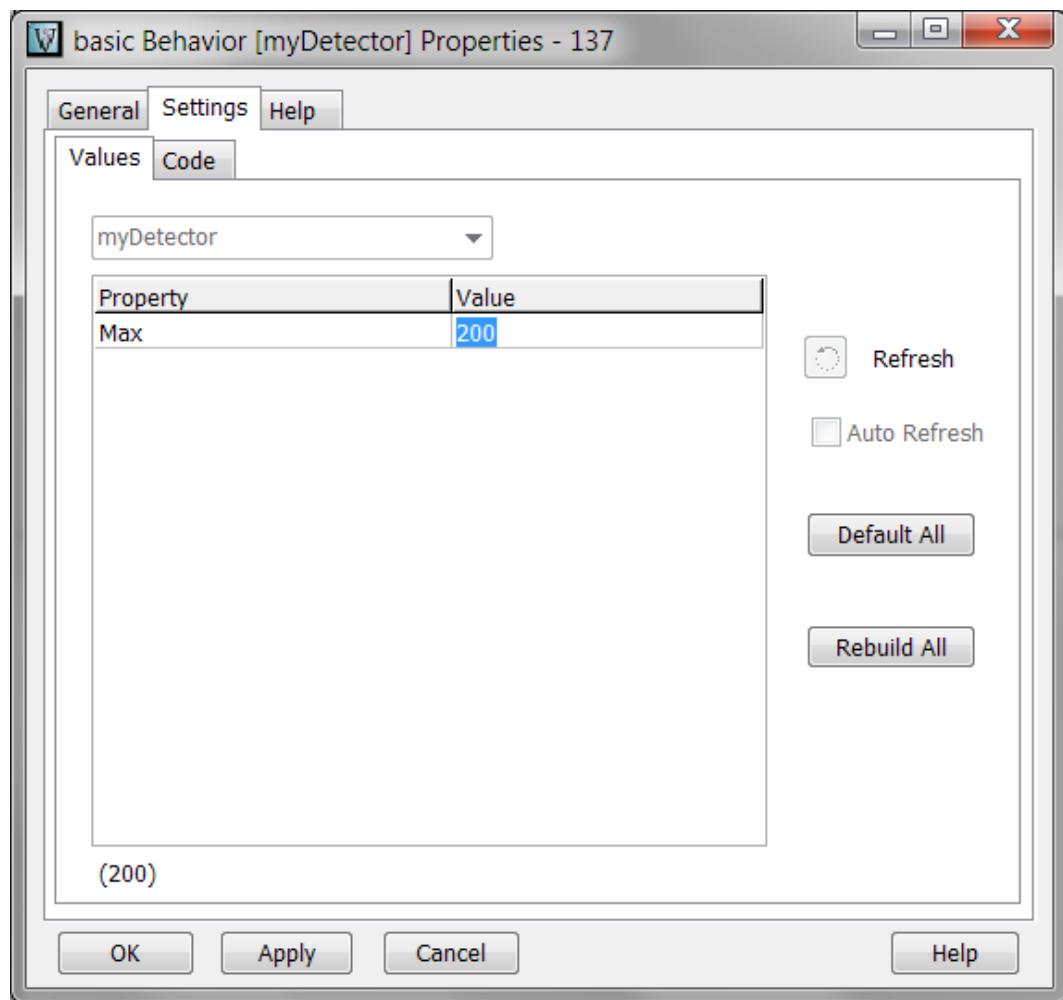


The Component (model) will be visible on the Diagram, with the others:



Double-click the Component (model) then, let's change the max value to 200

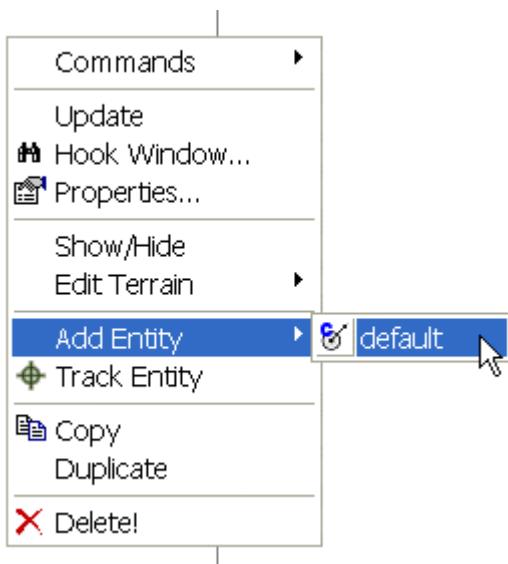
## Simple Component



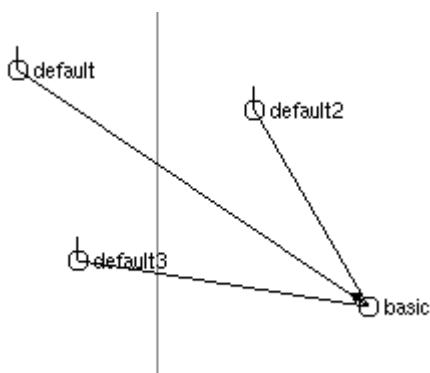
Set OK, then save, compile & run/start the simulation

Now, we need to add one (or more) entities to the scenario to check the `myDetector` model.

Let's do it during runtime by right-click the mouse like shown below:



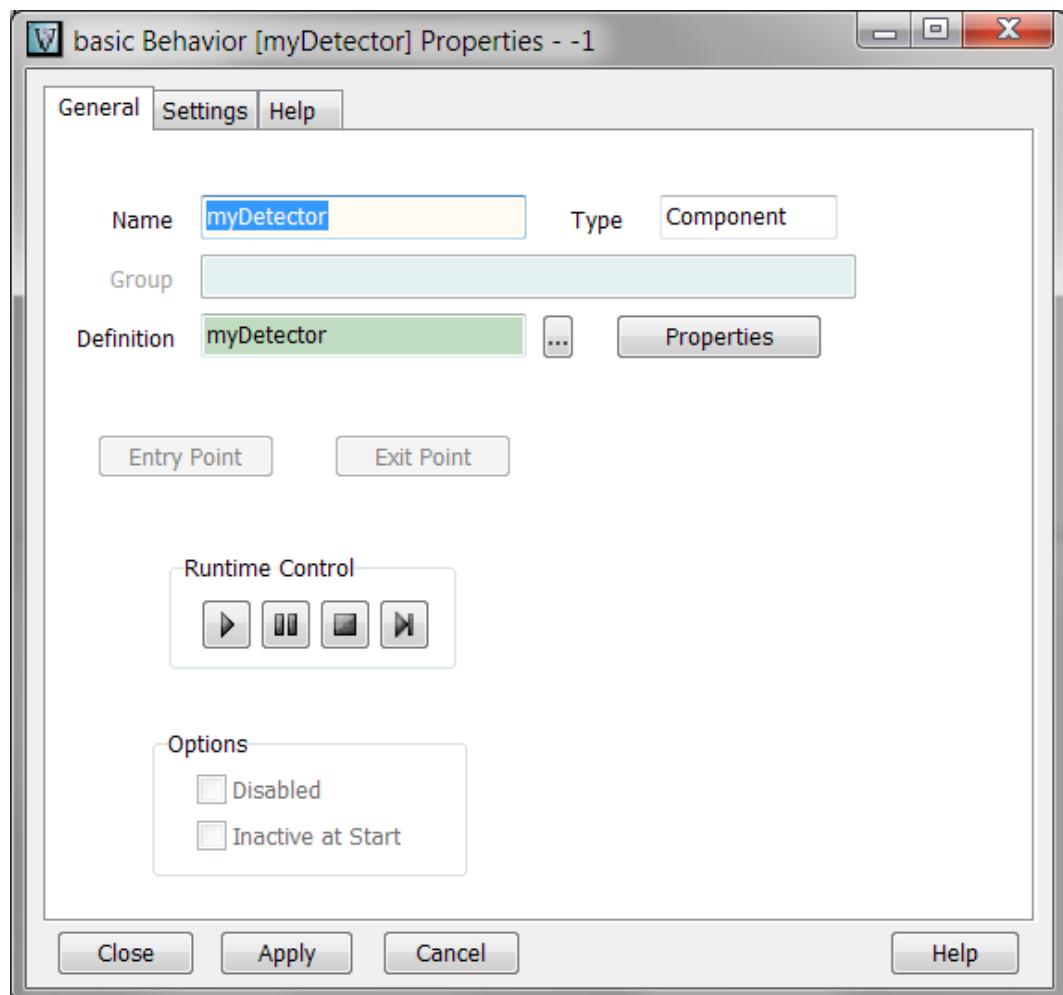
The new runtime entity will be added at the position of the mouse.  
If the added entity is close enough to the basic entity, a line will be shown. As soon as the distance get beyond 200, the line will disappears.



To finish with this simple demo, let's now turn off the Component during runtime.  
Select the basic Entity, then, the Model (behavior) pane, then double-click the [myDetector](#) component (shown in magenta because running) and click the stop button as shown below.

The lines will disappears as no detections are no more processed.

## Simple Component



# Sub-banding

Quite often, inside a component runtime, some checks do not need to be carried out at the component base rate. For example, checking the temperature of an engine component running at 30hz can be done every second or two.



*A sub-band can only by a division of the base component frequency. If the component is running at 30hz, a sub-band can run at 15hz or 7hz, etc*

The following example shows how to call a function, from the Runtime part of the component, at 2hz.

*Declaration:*

```
double sb_check; // to be set to 0 in the INIT part
```

*Runtime:*

```
sb_check += cycleT(); // cycleT() = theoretical time in second
between two calls of the component runtime
if (_check > 0.5) { doSomething(); sb_check = 0; } // 0.5 = 1/2
= 2hz
```

## Simple Knowledge

# Simple Knowledge

For this example, we will create a simple Knowledge that will constraint the basic entity of the previous example to zero the speed and heading when conditions are met.

First, we need to create a Knowledge.

Select Knowledge in the Environment Tree-List then, in the Display diagram section, right click and select Add Knowledge



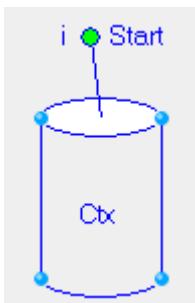
The Knowledge is created.

Double-click the Knowledge symbol (below) to Open its definition.

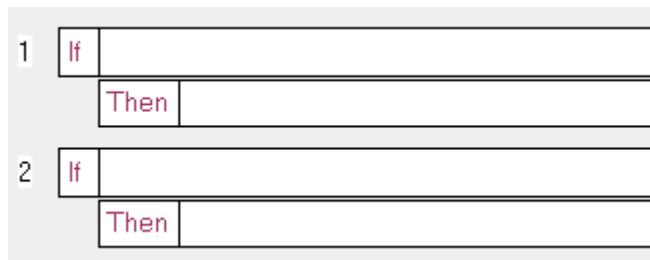


Now, you can use the Knowledge toolbar to create visually the Knowledge. In this case, we will create one Context including 2 Rules.

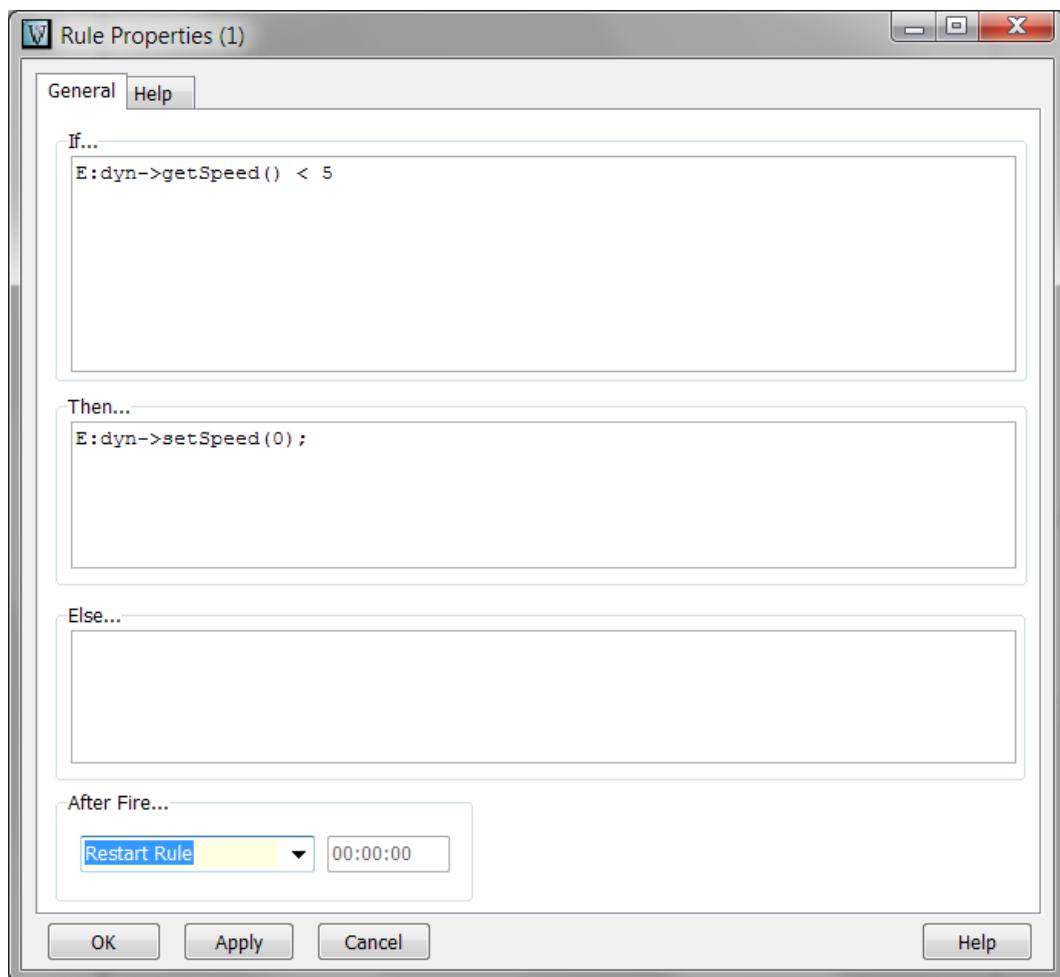
First, let's create the Context by double clicking the Knowledge **Knw**, then using the toolbar button before clicking on the Diagram background.



Double click the Context (above) then add two Rules by clicking two times the button.



Now, double click the first Rule representation on the diagram to define it:



Here, we specify that if the speed of the Entity is less than 5 m/s, then, the entity must stop (speed = 0).

After Fire, we restart the Rule to ensure it will be triggered again if needed.

Now, double click the second Rule representation on the diagram to define it:

## Simple Knowledge

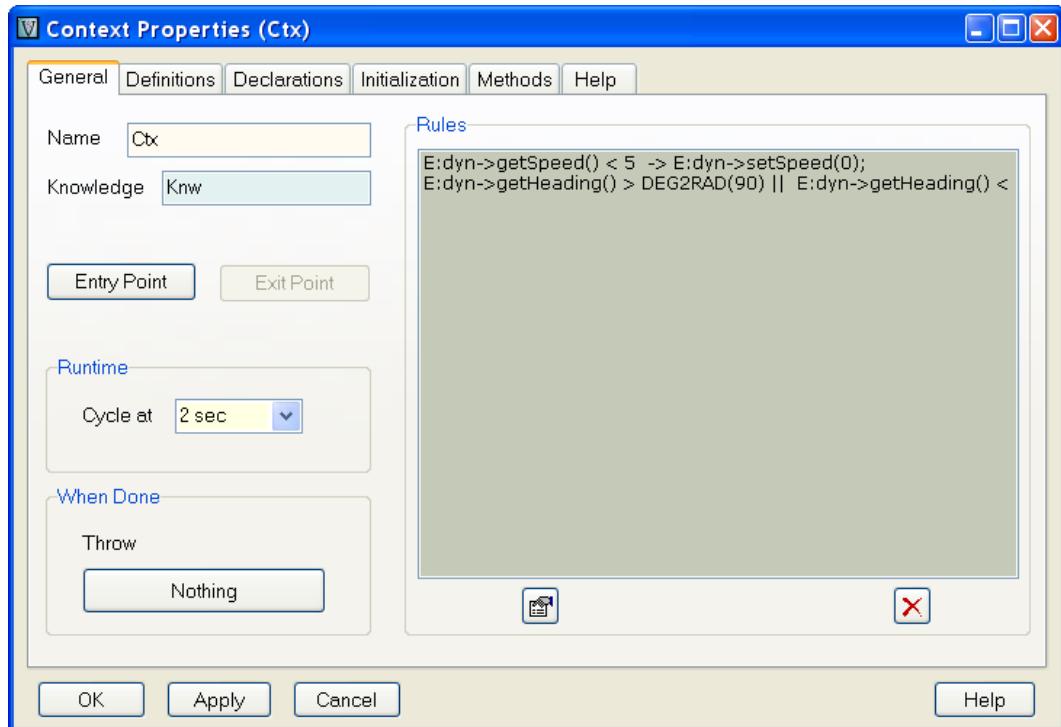
The screenshot shows the 'Simple Knowledge' editor interface. At the top, there are tabs for 'General' and 'Help'. Below the tabs, the 'If...' section contains the condition: E: dyn->getHeading() > DEG2RAD(90) || E: dyn->getHeading() < DEG2RAD(-90). The 'Then...' section contains the action: E: dyn->setHeading(0);. There is also an 'Else...' section which is currently empty. At the bottom, the 'After Fire...' section includes a dropdown menu set to 'Restart Rule' and a timer set to '00:00:00'.

In the second Rule, whenever the Entity heading is above |90| degrees, Entity is forced to go north (0).

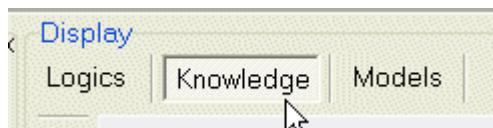
After Fire, we restart the Rule to ensure it will be triggered again if needed.

Finally, let's make the Context cycle every 2 seconds.





The Knowledge must be attached to the Entity (Behavior) the same way as we did for the Logic. Both will run in parallel.  
To do so, select the Entity (basic), then select Knowledge Pane in the Behavior.

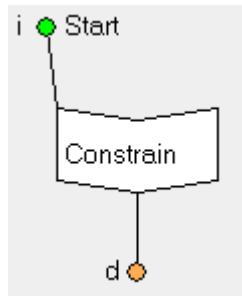


Let's then add the new **Knw** created Knowledge



And that's it (we can rename the Knowledge Behavior if we want)

### Simple Knowledge



Time to Save then Compile and Generate the code.

Start the Simulation.

Observe that the Entity basic will start moving randomly (speed and direction) from the Logic control but see how, every two seconds, the Knowledge will modify both speed & direction according to the two basic rules.

# **Simple HMI**

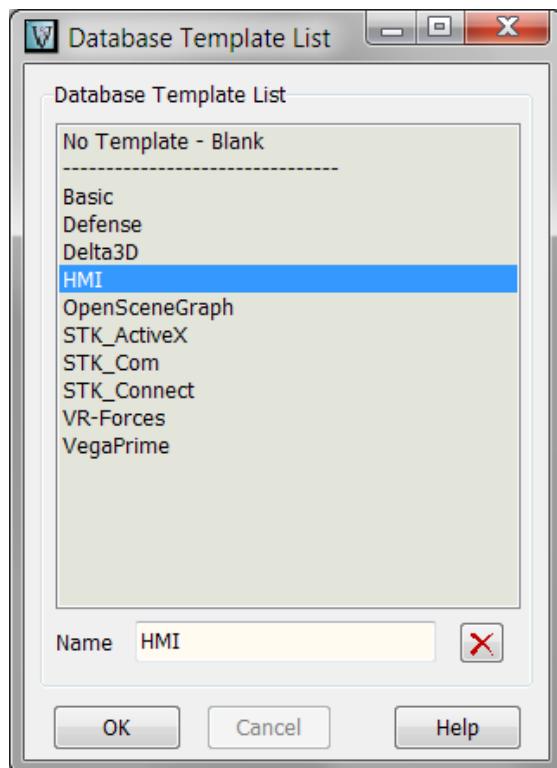
In this two samples, we will learn how to connect two sprites together and how to animate a sprite with a Component and a Logic

# **Input/Output Sprites**

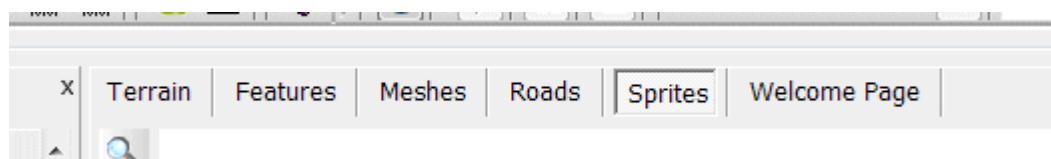
In this demo, we will add two sprites. One **input** Sprite object: **knob** and one **output** Sprite object: **lamp**

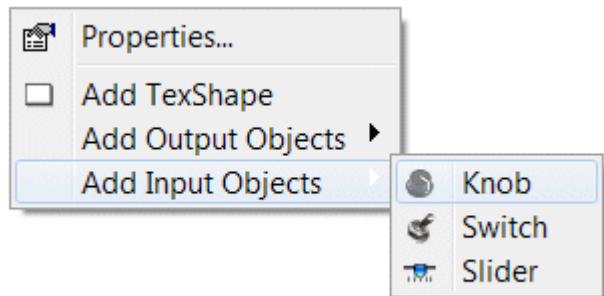
- **The Knob**

Let's create first a new Database by selecting the HMI template:

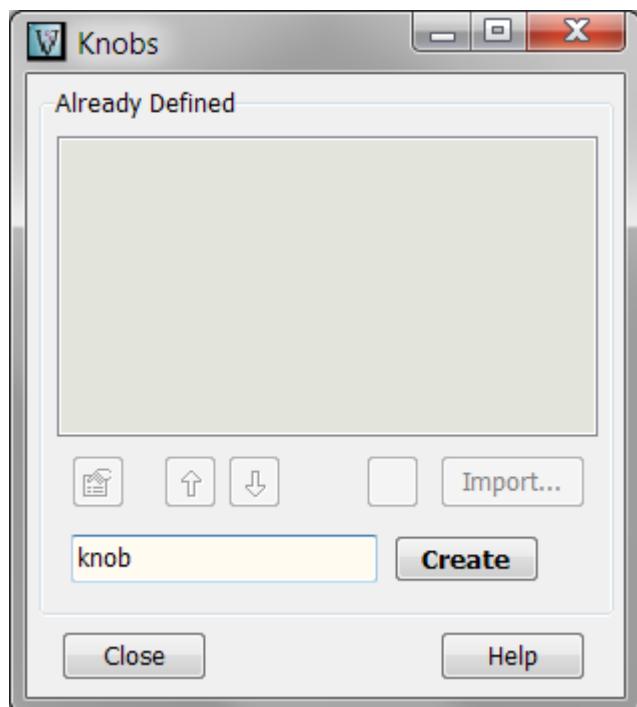


Now, let's select the Sprite pane and use the vertical toolbar to add the first input **knob** object:

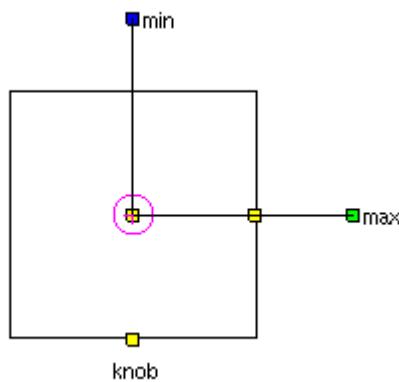




We can now call it **knob**:



press Create and click on the Area to set the Object:



## Input/Output Sprites

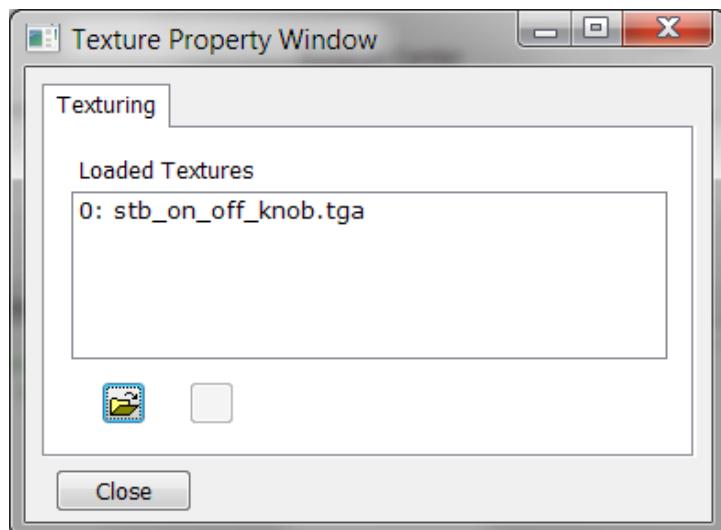
Let's open the Object properties by double-clicking it. We will load the moving texture by clicking [Visual Aspect](#) button then [Texture](#):

Let's select the `stb_on_off_knob.gif` texture:

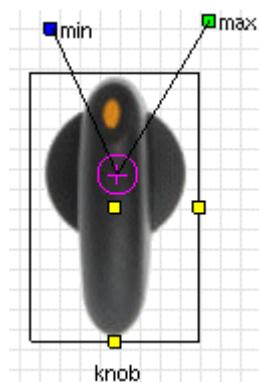


*vsTASKER uses TGA files for alpha channel transparency capabilities. But as windows does not display miniatures for TGA files, it is a good idea to have a GIF or BMP file with the same name.*

*At load, vsTASKER will check if a TGA extension file is available on the same directory and will use this file instead.*

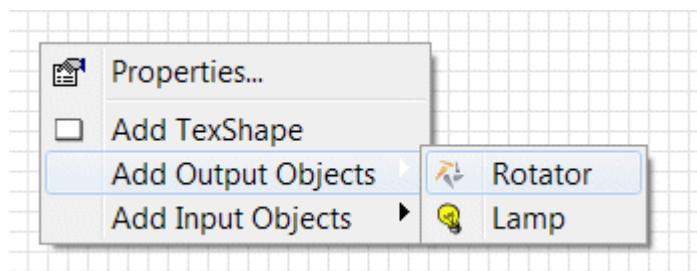


Offset the rotation center of the knob above the texture center as below and rotate the min and max handle like below:

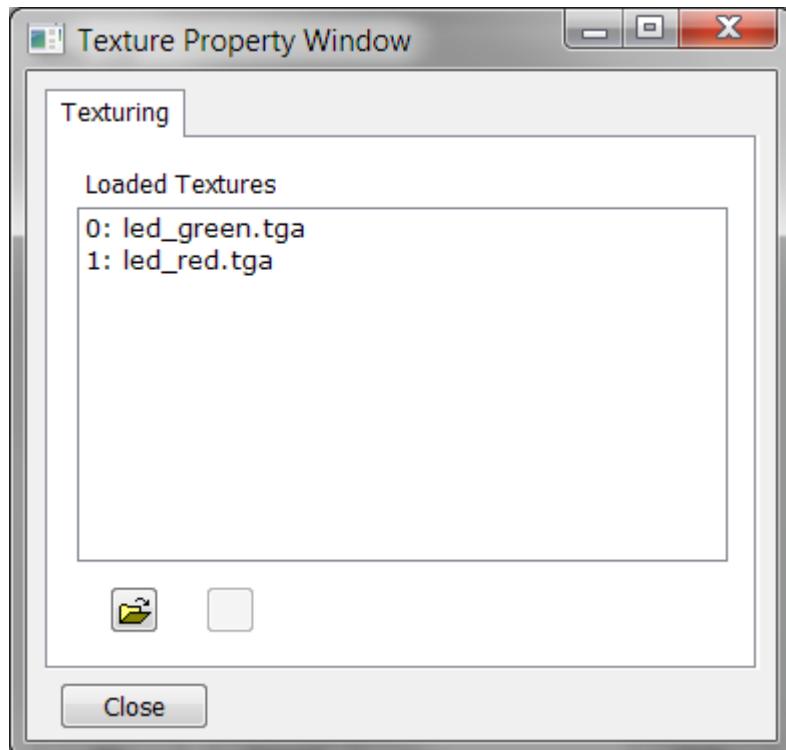


## • The Lamp

Now, let's add the Lamp Sprite:



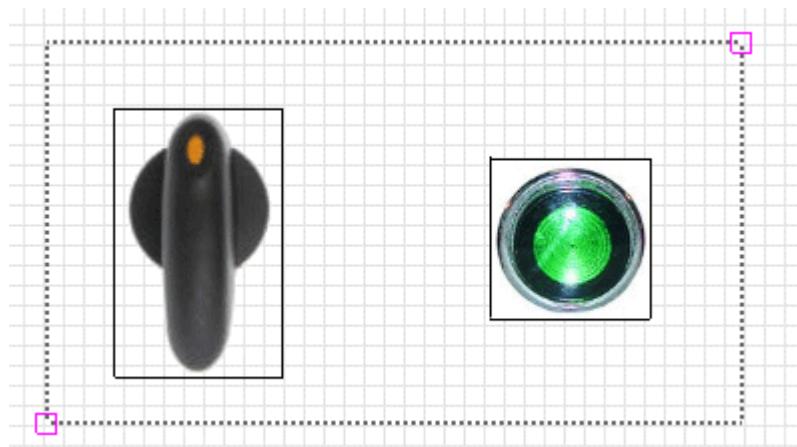
We need 2 states represented by 2 bitmaps: led\_green and led\_red:



*The number in front of the texture name is associated with the state of the lamp.*

Once the lamp is created, set its position right to the knob, then click the runtime window area dashed line and move the square handle to enclose the two Sprites as below:

## Input/Output Sprites



### • Coding the interaction

Now, we need to make the input knob activate the output lamp.  
For that, we open the knob property window and we set the initial value and the runtime action:

Definitions   Initialization   Runtime

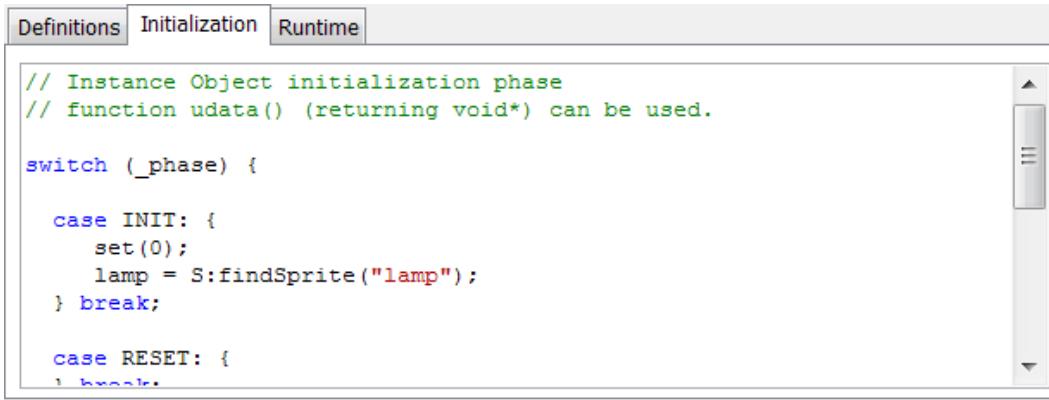
```
// Define here your Object local data

public:

private:
    Vt_Sprite* lamp;
```

We need to hold the lamp pointer as the knob is going to activate it.  
So, we define a `Vt_Sprite` pointer named `lamp`, in the **Definition** pane.

## Input/Output Sprites



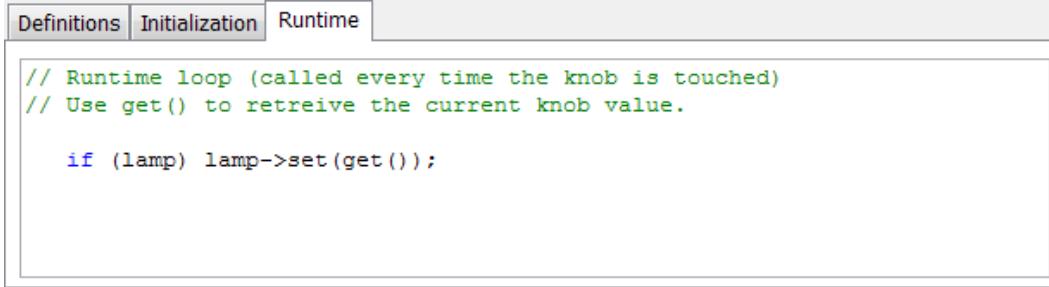
The screenshot shows the vsTASKER IDE interface with the 'Initialization' tab selected. The code in the editor is:

```
// Instance Object initialization phase
// function udata() (returning void*) can be used.

switch (_phase) {
    case INIT: {
        set(0);
        lamp = S:findSprite("lamp");
    } break;

    case RESET: {
        ...
    }
}
```

In the **Initialization** pane, we set the **knob** state to be 0: `set(0);`  
Then, we find the **lamp** pointer using the scenario method: `lamp = S:findSprite("lamp");`



The screenshot shows the vsTASKER IDE interface with the 'Runtime' tab selected. The code in the editor is:

```
// Runtime loop (called every time the knob is touched)
// Use get() to retreive the current knob value.

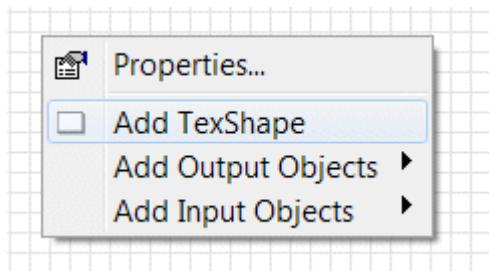
if (lamp) lamp->set(get());
```

The **Runtime** part is called every time the **knob** is touched or position changed.  
Here, we set the **lamp** state according to the **knob** state: `lamp->set(get());`

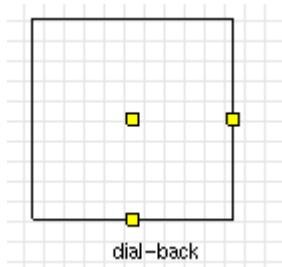
## Sprite Model

In this demo, we will create a speed dial and attach a Component model to it. First we create the dial by combining a TexShape and a Rotator:

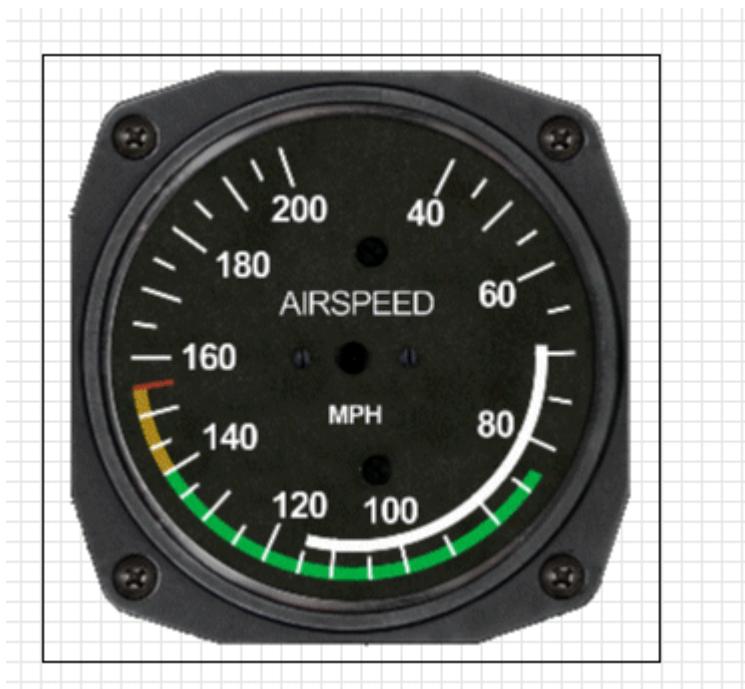
- **Building the Speed Dial**



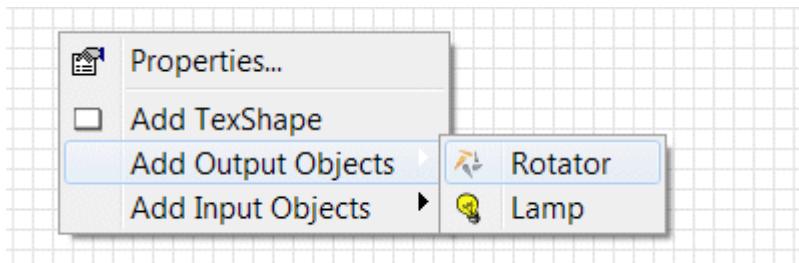
Let's call it: dial\_back



We we load the texture air\_panel. Do not forget to click the  Auto Size check-box



Now, we will overlay the rotating needle. We will add a Rotator Sprite we name [dial\\_needle](#)



Then, we load the [air\\_needle](#) texture in Visual Aspect (do not forget to check the [Auto Size](#)  box)

We can now move the [dial\\_needle](#) over the [dial\\_back](#), set the rotation center at the [dial\\_back](#) center and set the minimum angle (blue handle) to 0 and maximum angle (green handle) to 200:

## Sprite Model



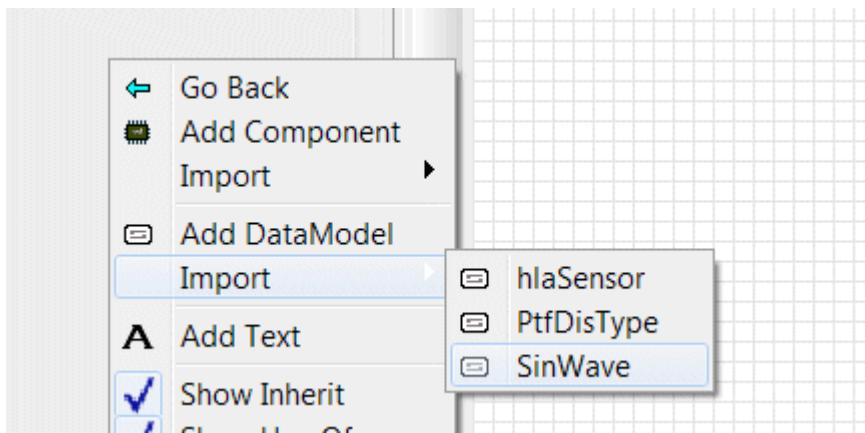
In the property window, we will specify the real values associated with the needle angles (`0 -> 0` and `-19 -> 200`):



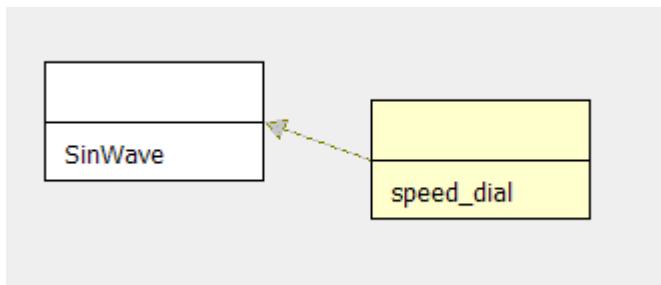
## • Making the runtime Component

Now, let's create a Component that will make the needle go from 0 to 200 and back continuously.

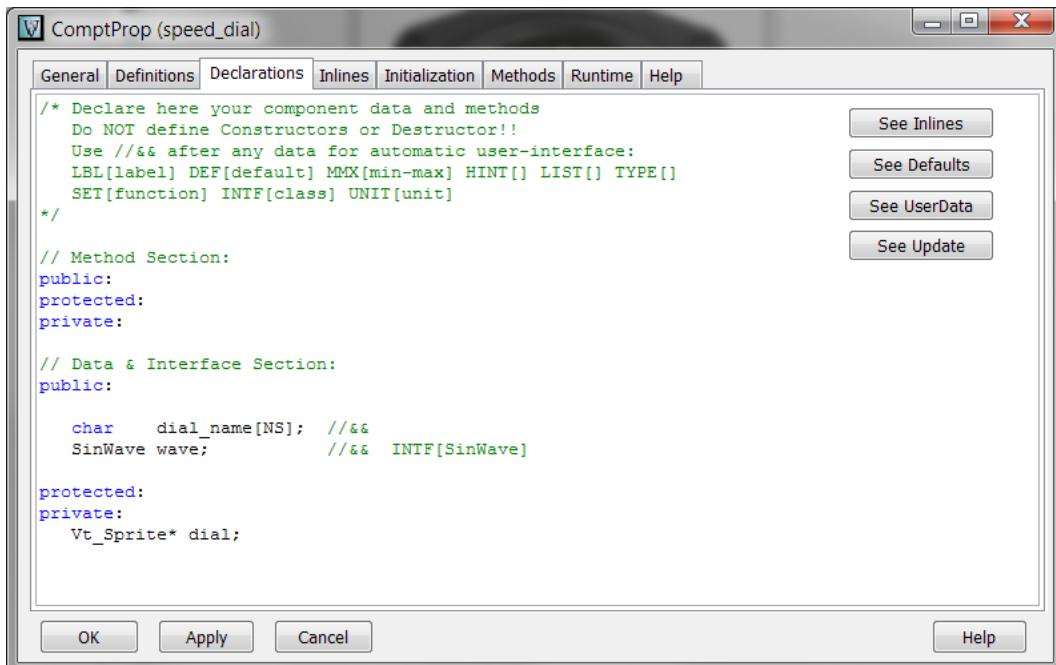
In the Model section, add the [SinWave](#) DataModel by importing it from the external list:



Now, create a new Component named `speed_dial` that will use `SinWave` (this is only for correct `#include` purposes at code generation)



In the `speed_dial` component **Declarations** section, let's add the following:



## Sprite Model

```
char dial_name[NS]; // so that the user can specify which object  
to rotate  
SinWave wave; // to tune the bounds and the speed  
Vt_Sprite* dial; // to access the Rotator Sprite in Runtime  
section
```

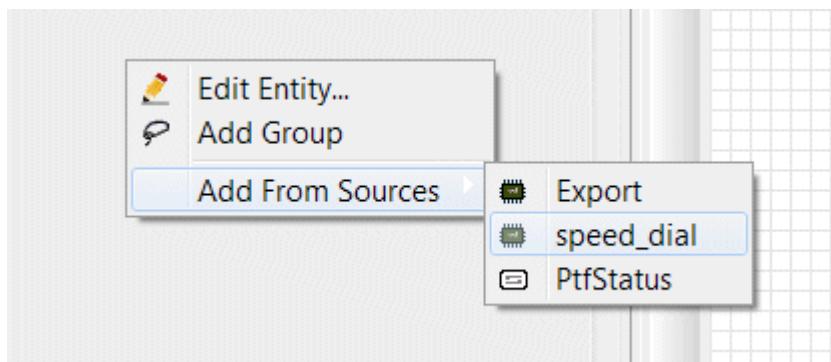
In **Initialization** section, we will just do the following:

```
case RESET: { // at each restart  
    dial = S:findSprite(dial_name);  
    wave.reset();  
} break;
```

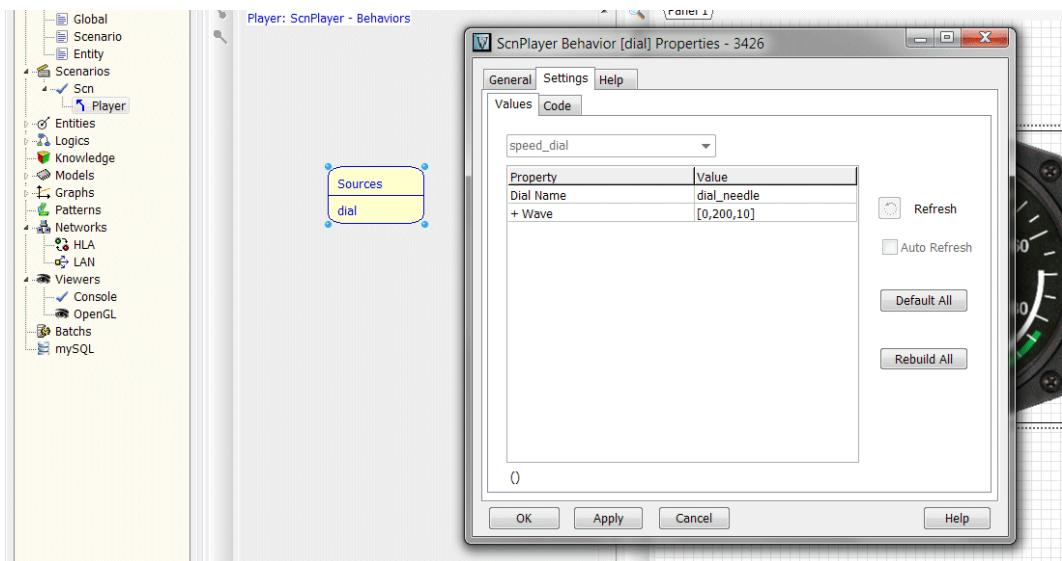
In the **Runtime** section, we will set the Rotator value with the one given by the SinWave DataModel

```
if (dial) {  
    dial->set(wave.get());  
}
```

Now, we can attach the **speed\_dial** component to the Scenario Player and setup the SinWave parameters:

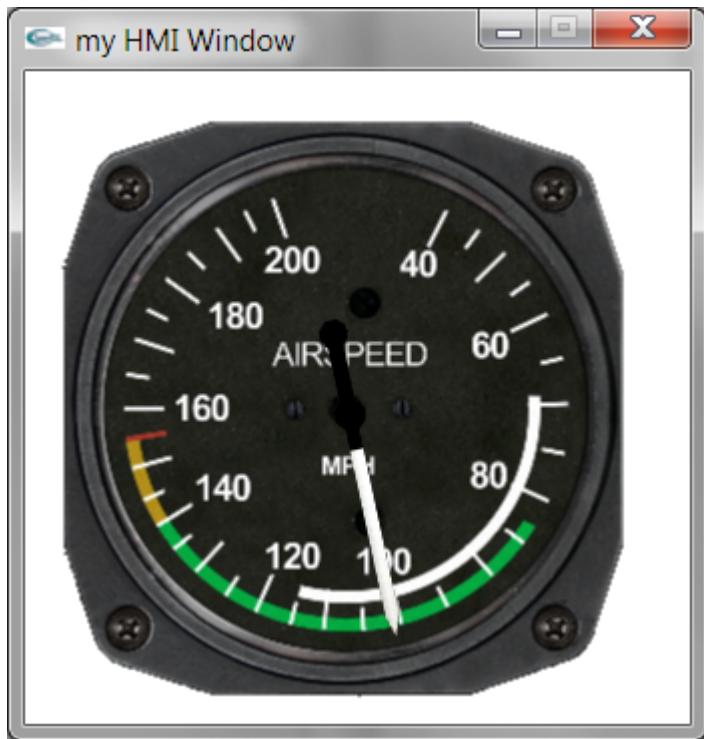


## Sprite Model



Compile, load and run the simulation.

You should get a Glut Window that shows a Speed Dial moving from 0 to 200 smoothly and continuously (making the RTC running at 60 hz will increase the smoothness)



## **Using OpenSceneGraph**

# **Using OpenSceneGraph**

<TODO>: Insert description text here... And don't forget to add keyword for this topic

# **Going Further**

**DIS**

# **DIS**

The chapter will introduce the use of vsTASKER with DIS protocol.

# HLA

The chapter will introduce the use of vsTASKER in an HLA environment.



*Specific license is mandatory. Contact vendor if you need this module.*

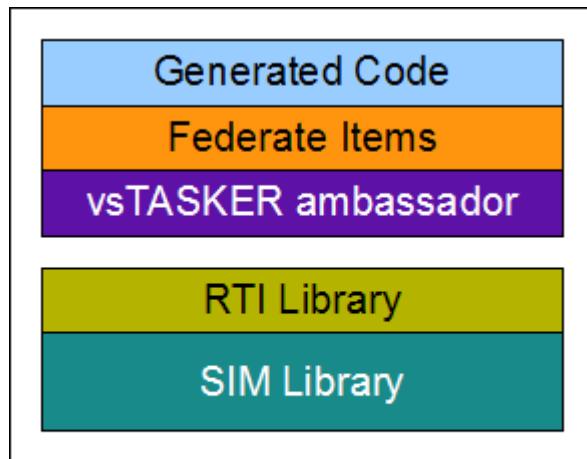
# Concepts

vsTASKER will generate the code that combines the following items:

- SOM objects, including the publishing and subscribing mechanisms
- Ambassador callback accesses

For that reason, it is important to understand what is generated, what is mandatory for the user to do and how things organize.

The following picture roughly describes the content of an HLA simulation engine produced by vsTASKER:



Simulation Engine

## • Base Class

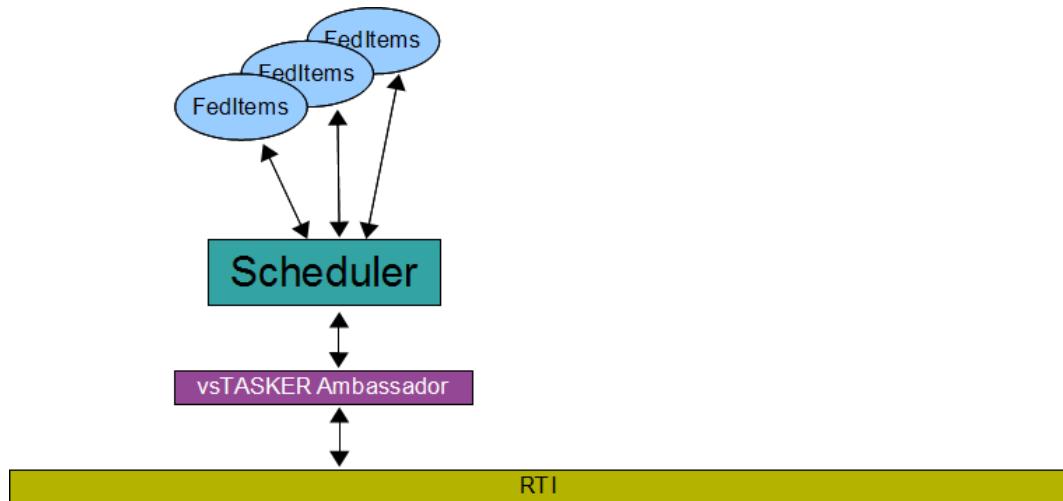
vsTASKER provides a way to connect to the RTI agent by use of a class (`vsTaskerAmbassador`) that inherits from `BaseFederateAmbassador` which provides the standard API to the HLA agent, as described by the HLA norm (vsTASKER chooses the correct `BaseFederateAmbassador` class from `include/hla` according to the RTI provider and version selected by the user).

The `vsTaskerAmbassador` class is automatically generated and overwrite all callbacks to connect to the federation, setup all services, publish and subscribe and manage the RTI data flow.

The class also contains all items (see below) defined by the user.

When the simulation starts, one instance of the `vsTaskerAmbassador` class is created.

It joins the Federation (or create it) and is seen as a named vsTASKER federate. Behind this federates, all the items are running.



## • Federate Items

From the outside, a vsTASKER simulation engine that is connected to the RTI is seen as a Federate.

From inside, the federate contains several items called Federate Items or **FedItems**.

These **FedItems** can be either Object or Interactions.

They can publish or/and subscribe.

Each **FedItem** is associated to a SOM.

The user must add the code to read or write the data for each attribute/parameter or the object/interaction associated with the **FedItem**.

vSTASKER provides some preprocessing capabilities to simplify the data transfer but cannot do all.

It is to the user to know what is the data that is carried thru the RTI and how to convert and use it.

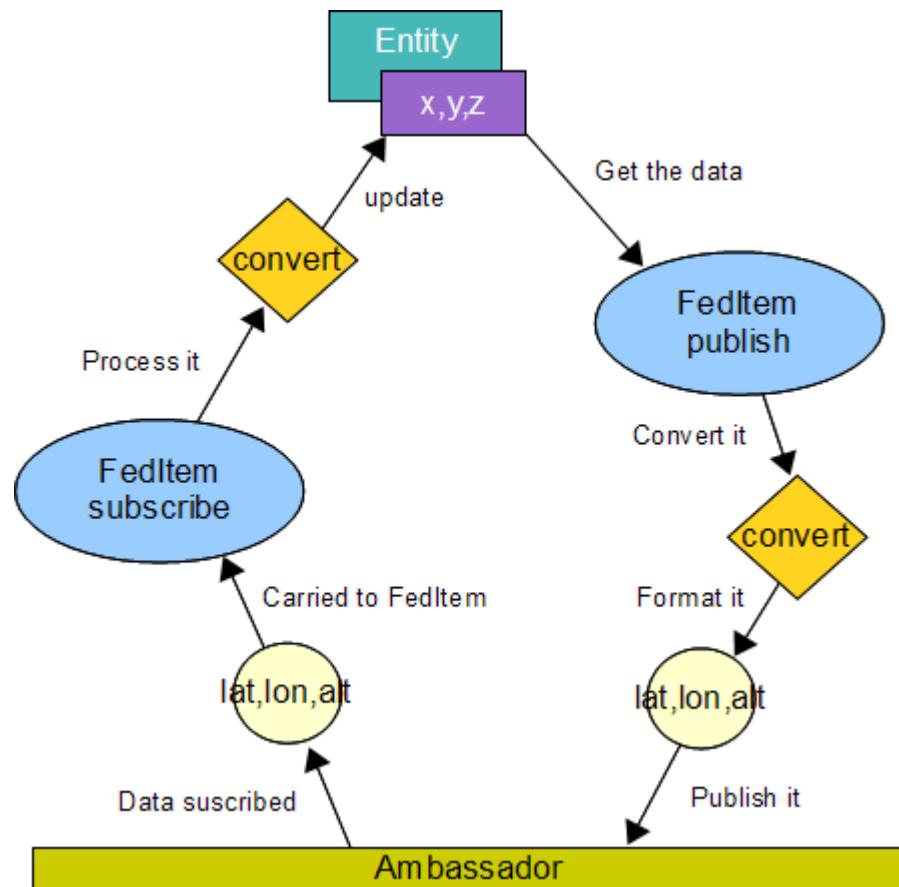
The **FedItem** can also be associated with a **DataModel** that can exactly match the SOM definition, in order to reduce the data manipulation and storage.

A **FedItem** Object must be seen as a class that will create as many contexts as necessary, according to the number of real users.

## Concepts

For example, a **FedItem** Object that will publish the position (x,y,z) and the name (string) of an entity will create one context for each scenario entity that will use it, either to publish their positions to the Federation, or to update their position from a remove federate.

It will then be to the user to make sure that the **FedItem** will correctly get the data from the internal structures of the entity and publish or update them.



# Simple Example

- **Going HLA**

We will now see step by step how to create a simple simulation that will make two vsTASKER (Simulation Engine) communicate.

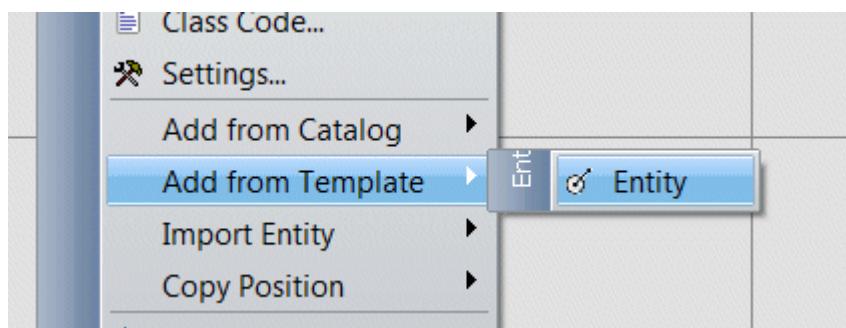
Both will have one entity and will intend to see other federate entities.

For this, we are going to use the vsTasker Federation included.

This Federation is quite simple and used to make several vsTASKER to share a simulation.

Open vsTASKER, create a new database from [Template Basic](#).

Now, let's put an Entity on the terrain.



Let's give the entity a heading (45) and a speed (5)

Save database under name [simple\\_hla](#)



*When creating a new database, it is mandatory to save it quite soon under a name. Several functions are not available with a fresh unsaved database (popup menus, import/export, etc.).*

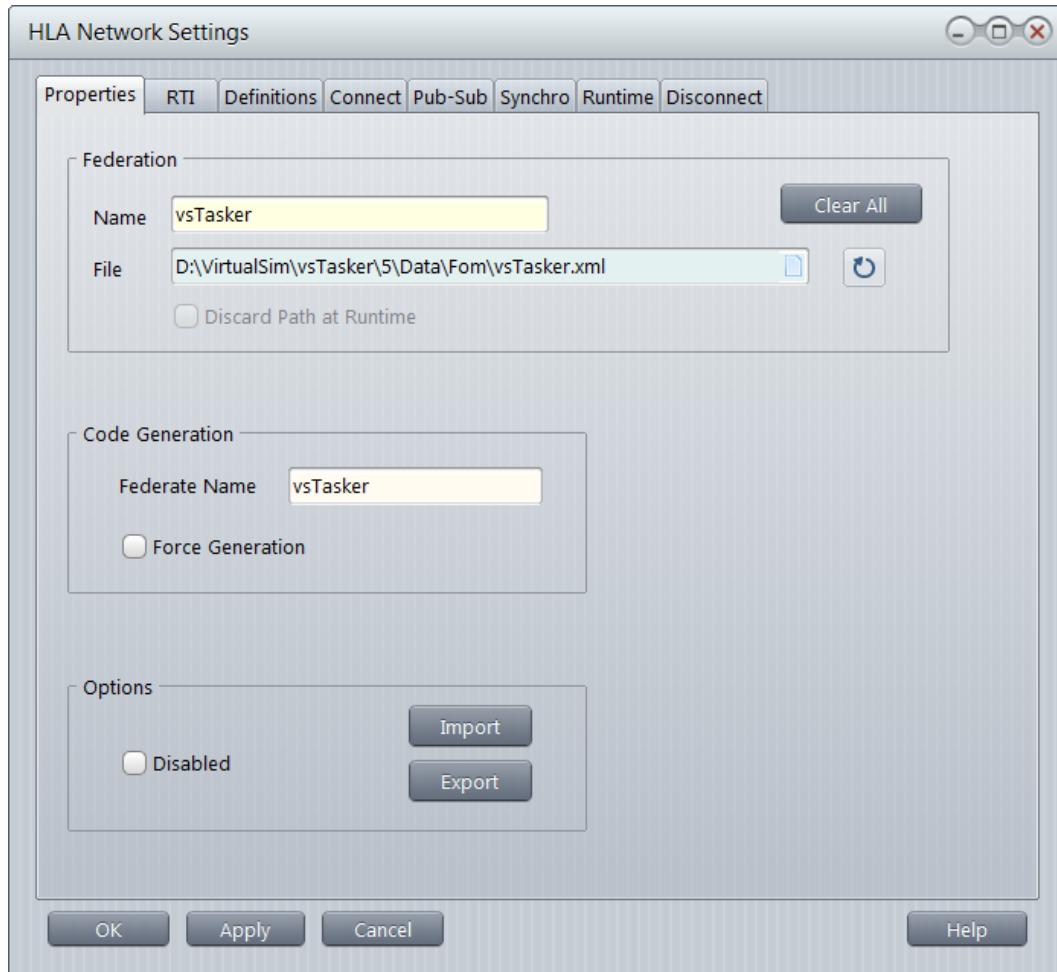
## Setting the RTI

# Setting the RTI

In Environment, click on Networks then HLA.

We will first open the HLA settings window by double clicking the HLA diagram part (or right click, then Settings)

Then we select the file: **vsTasker.xml**

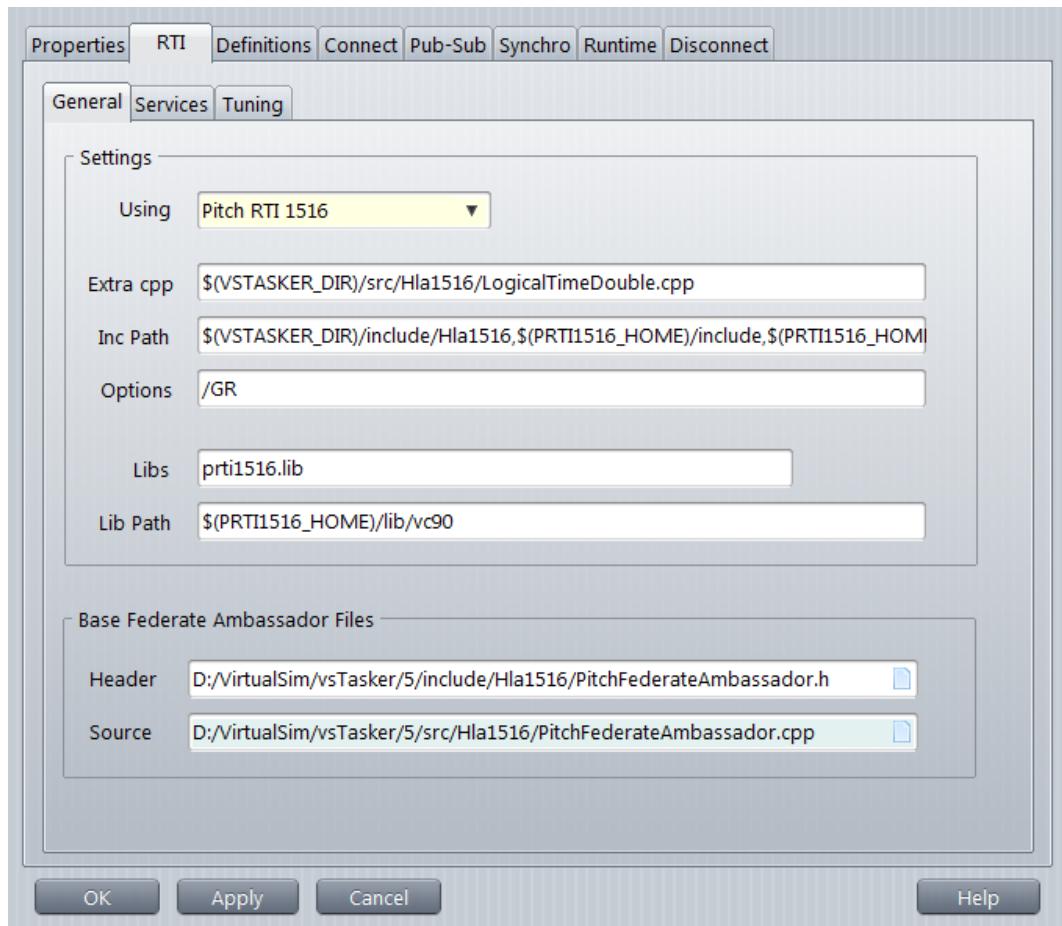


Now, let's select the RTI to use.

If you have Mak RTI, select [Mak 1516](#) (or Evolved)

If you have Pitch RTI, select [Pitch 1516](#) (or Evolved)

## Setting the RTI



For now, it done for the RTI settings.

# Mäk RTI

vsTASKER has been tested with **Mäk RTI 4.3**. Previous versions should also work. Make sure that you have installed the correct Visual Studio version for the RTI and that it will be the one you will use with vsTASKER.

If samples does not compile or crash, please to inform us at:  
[support@virtualsim.com](mailto:support@virtualsim.com)

## • Environment Variable

Make sure you have the following variable setup and pointing to the Mäk RTI installation directory:

```
MAK_RTIDIR -> C:\MAK\makRti4.3 (for ie)
```

Path environment variable should include the following:

```
%MAK_RTIDIR%\bin
```

## • Handle File

It is also important to alter the 1516 and 1516e **handle.h** file this way, otherwise, samples won't compile:

```
public: \ <- add this line just before the _impl declaration (below)  
HandleKind##Implementation* _impl;
```

# Pitch RTI

vsTASKER has been tested with **Pitch RTI 5.3**. Previous versions should also work. For newest versions, if samples does not compile or crash, please to inform us at: [support@virtualsim.com](mailto:support@virtualsim.com)

- **Environment Variable**

Make sure you have the following variable setup and pointing to the Pitch RTI installation directory:

```
PITCH_RTIDIR -> C:\Pitch\prt1516e (for ie)
```

Path environment variable should include the following (assuming you want to use Visual Studio 2010)

```
%PITCH_RTIDIR%\lib\vc100
```

- **Handle File**

It is also important to alter the 1516 and 1516e **handle.h** file this way, otherwise, samples won't compile:

For HLA1516-2000:

```
public: <- replace the private with public just before the _impl
declaration (below)
    HandleKind##Implementation* _impl;
```

For others:

```
public: \ <- add this line just before the _impl declaration (below)
    HandleKind##Implementation* _impl;
```

## Defining SOM

# Defining SOM

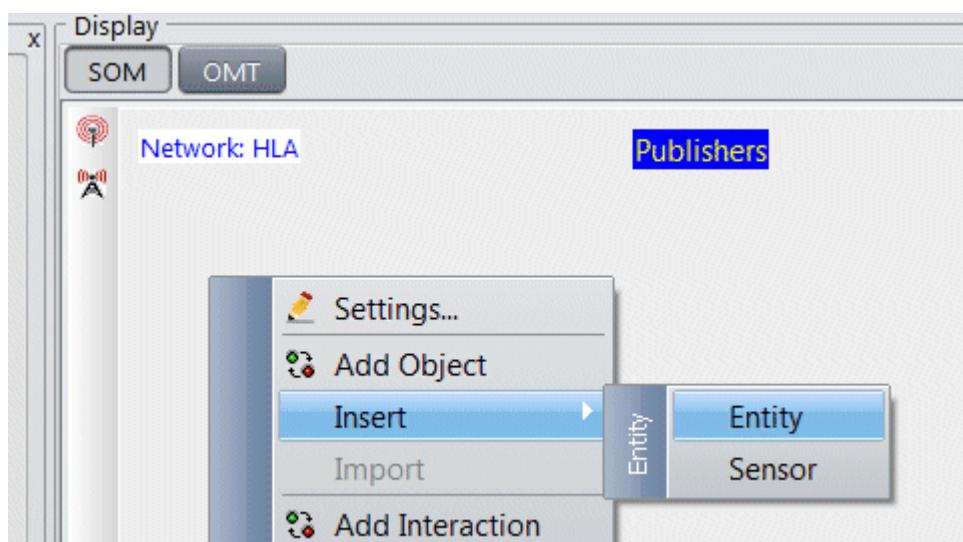
Now, it is time to define the SOM by adding all Objects and Interactions that will participate in this Federation.

We want to Publish and Subscribe the position only of an Entity.

We just need two [FedItems](#).

One to Publish and one to Subscribe.

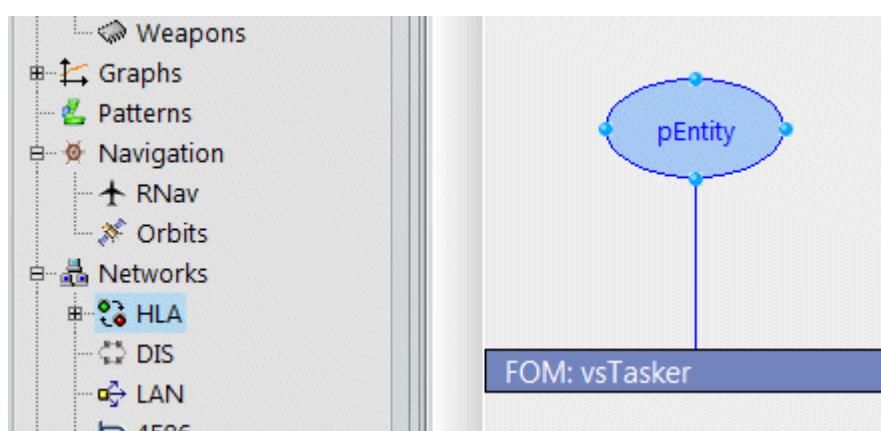
We will add first the publishing [FedItem](#) by right clicking into the Publishers part of the HLA diagram.



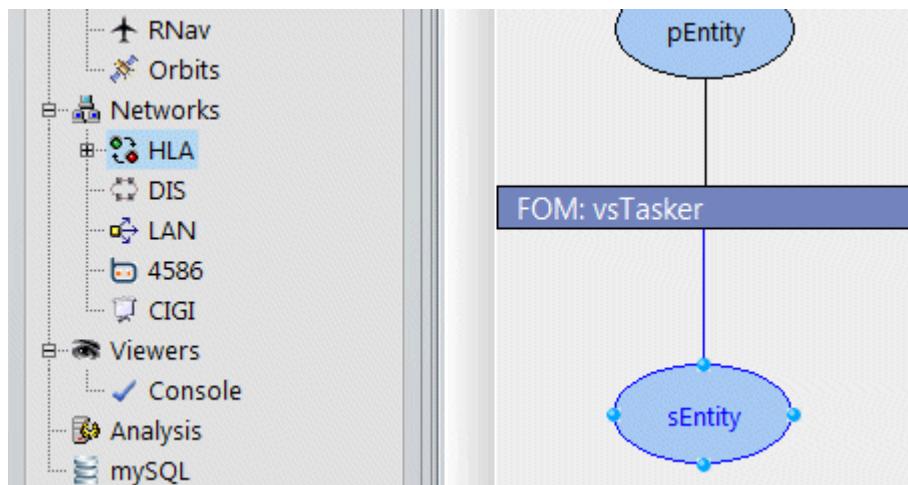
The publishing [FedItem](#) is then created.

You can select it and drag it wherever you want on its upper area (even resize it if needed).

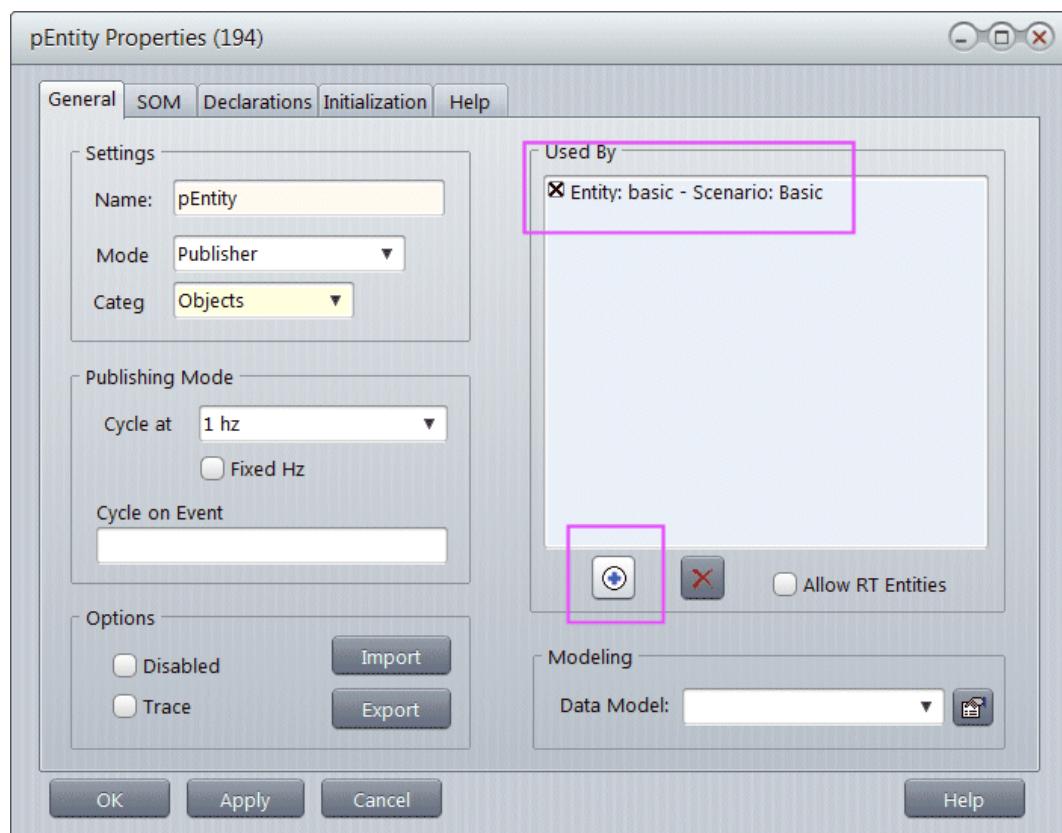
Use the button to revert to the default size.



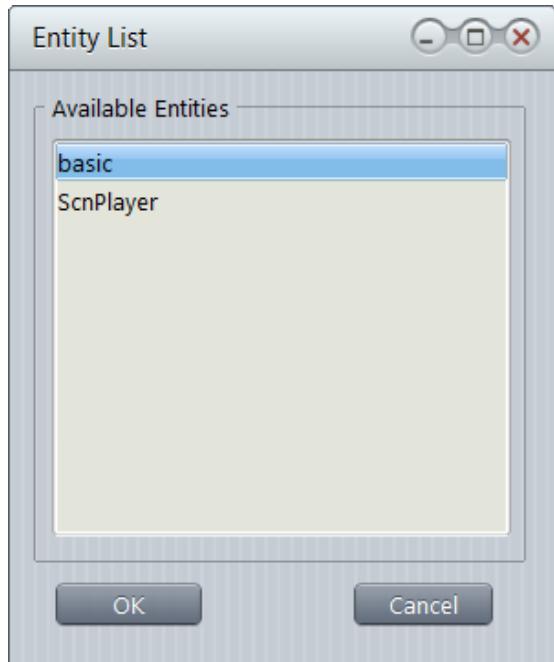
Let's do the same with the Subscriber



Finally, attach the entity to the Publisher [FeedItem](#), because the entity *basic* will be using it to broadcast its position, heading and speed through the RTI.



## Defining SOM



We will use two methods to get/set the data.

It will be up to the user to decide which one he prefers although the DataModel one is more suitable for large Federations.

The first one will use [Direct Access](#) (to the data)

The second one will use [Data Models](#)

## **Using Direct Access**

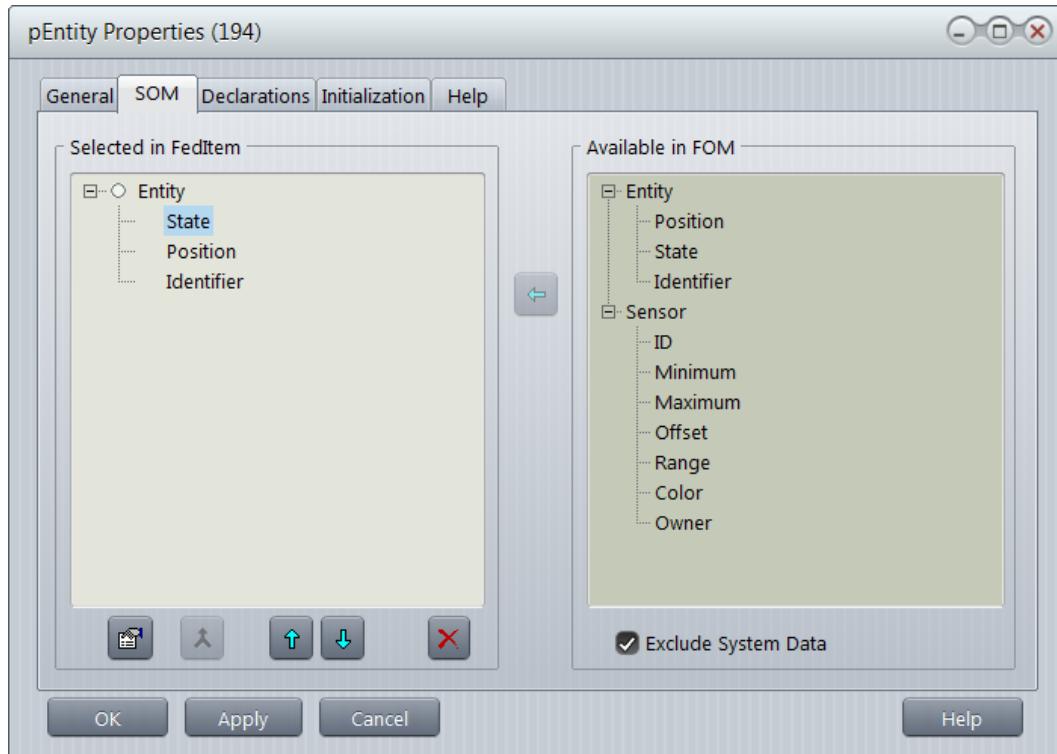
This mode can be used for RTI 1.3, with a simple Federation or to understand the mechanism behind the [FedItem](#) object.  
It requires a little more coding.

## Publish

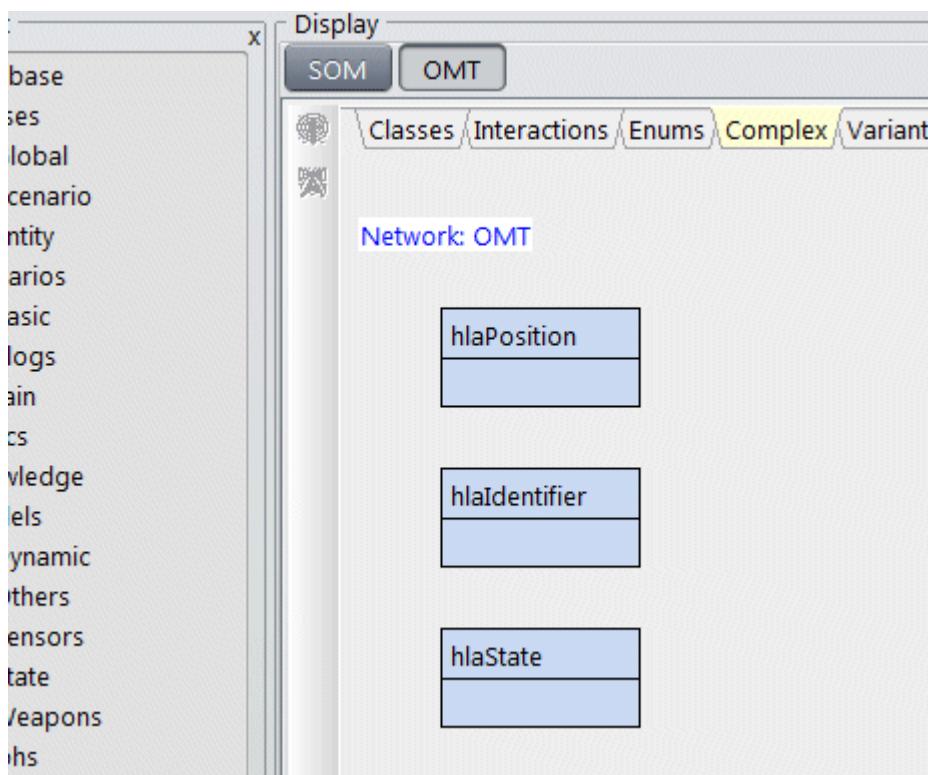
# Publish

- **Defining the Data**

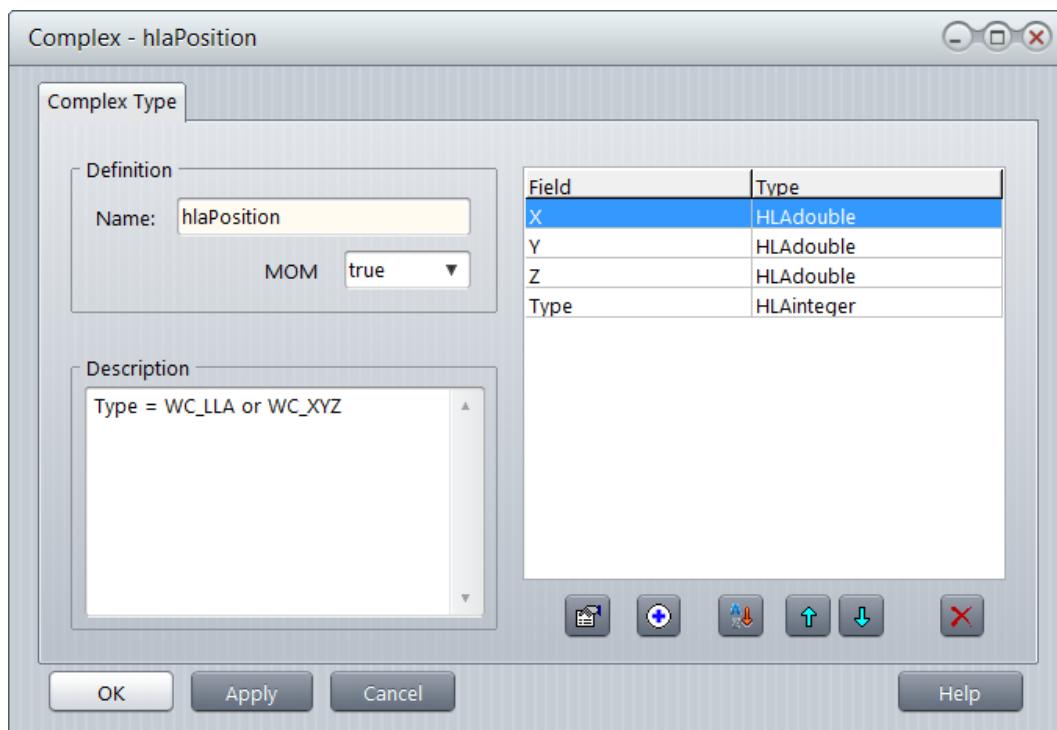
Now, double-click on the **publishing** FedItem to call its Property Window



Here, we can see that Entity Object has been selected and that the three Attributes ([State](#), [Position](#) and [Identifier](#)) must be mapped somehow with some local variables. When a Federation (1516 XML) file is loaded, vsTASKER generates an OMT like definitions that can be viewed on the OMT part.



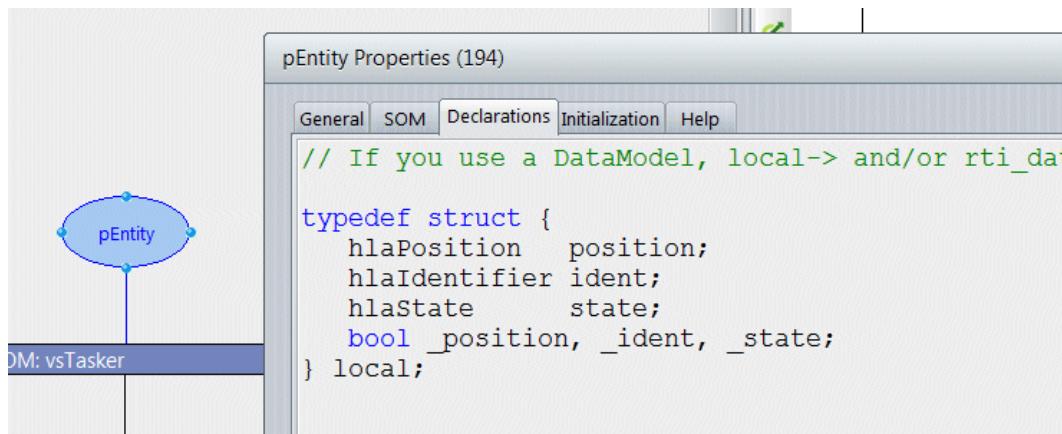
Now, let's see what [hlaPosition](#) contains, but opening the symbol:



## Publish

vsTASKER will automatically generate OMT Structs and Enums in the header file, so, it is useless to recreate them (as it will generate compilation errors). So, we know that the header will contain the following structure definition:

```
typedef struct {
    HLAdouble X;
    HLAdouble Y;
    HLAdouble Z;
    HLAinteger Type;
} hlaPosition;
```



```
typedef struct {
    hlaPosition position;
    hlaIdentifier ident;
    hlaState state;
    bool _position, _ident, _state;
} local;
```

## • Setting the Attribute

We know that every time an [Object Attribute](#) is modified on the Publisher side, the RTI will automatically update all corresponding Subscriber's [Object Attribute](#). So, we need to publish the [Attribute](#) whenever it has been modified by the [Object](#).

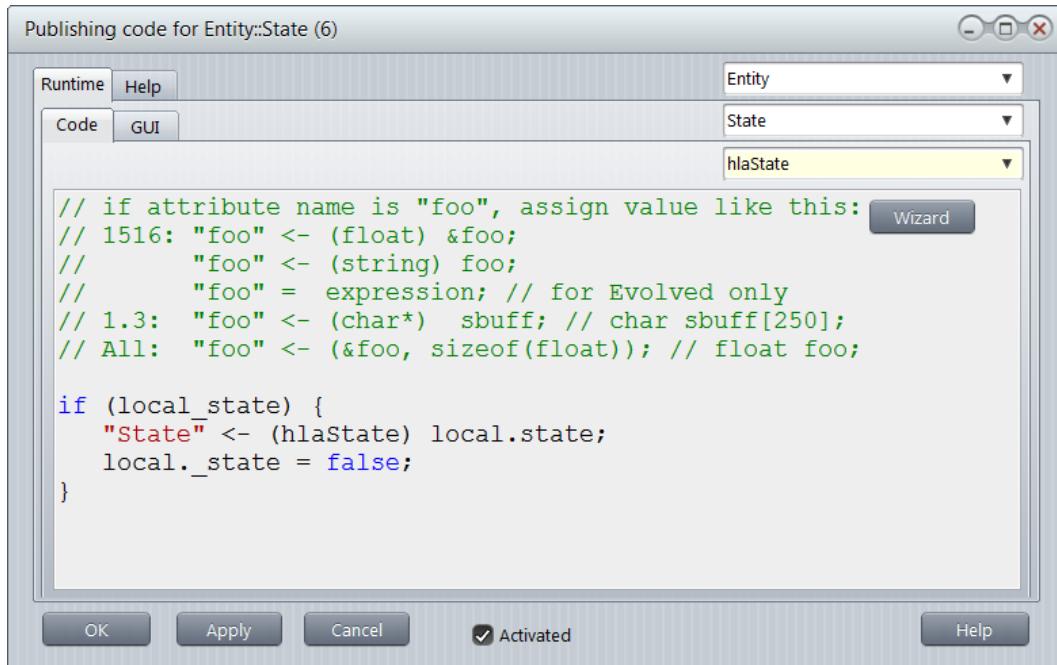
In the FedItem property window, let's select SOM panel and let's double-click the [State](#) Attribute.

In the Code section, let's add:

```
if (local_state) {
    "State" <- (hlaState) local.state;
```

```
    local._state = false;
}
```

vsTASKER will pre-process this part and translate it into HLA API C++ code.  
 "State" must be understood as "the RTI data counterpart".



Let's do it for the three [Attributes](#) (on each respective window) :

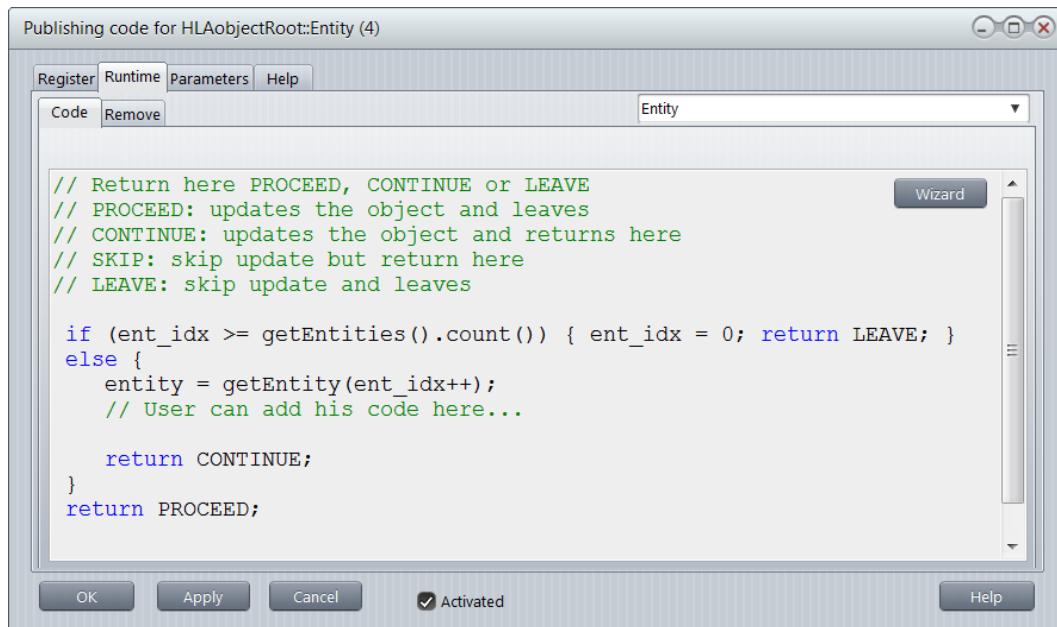
```
if (local._position) {
  "Position" <- (hlaPosition) local._position;
  local._position = false;
}

if (local._ident) {
  "Identifier" <- (hlaIdentifier) local._ident;
  local._ident = false;
}
```

## • Setting the Object

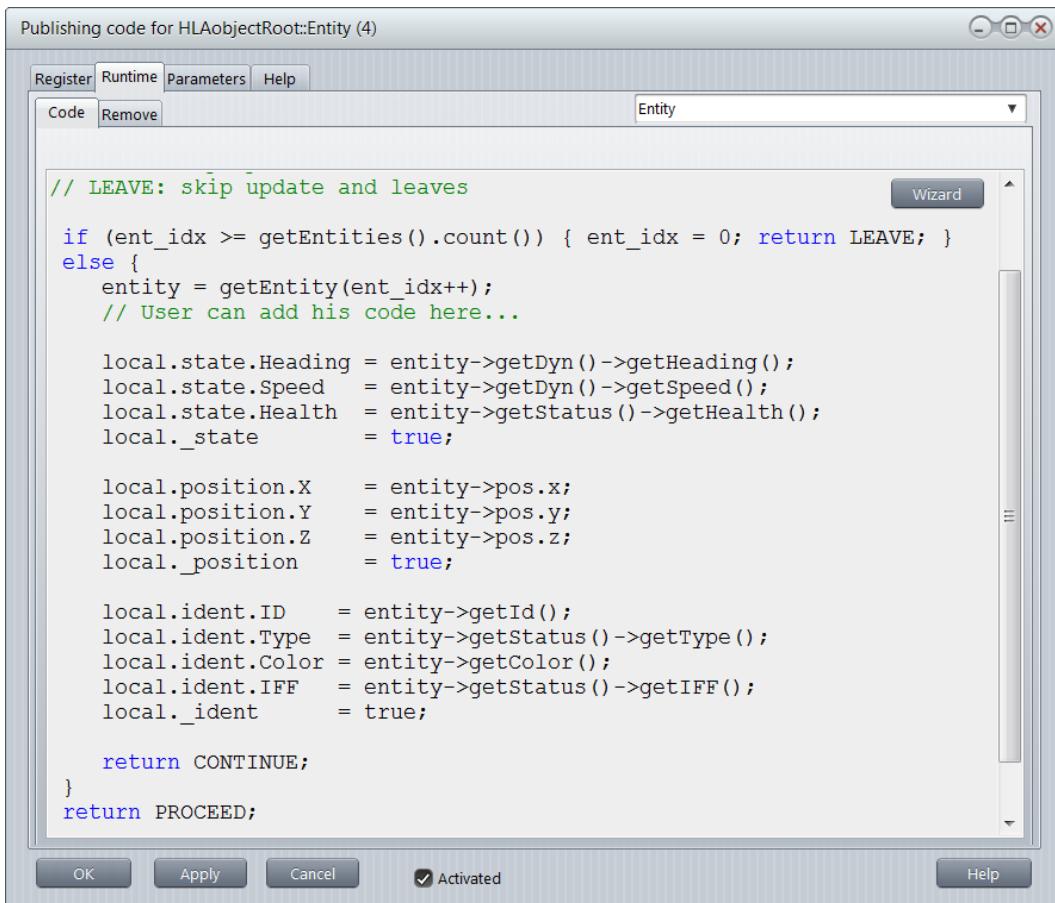
Now, we must update the Attributes for each FedItem users.  
 For that, open the [FedItem](#) property window at the [Runtime::Code](#) panel:

## Publish



This part of the code is called at the frequency set for the FedItem.  
Each registered entity (user) attached to the FedItem (see used\_by list) will be used.

`ent_idx` will parse all indexes from 0 to n-1 entities using this FedItem.  
We will put our update code as follow:



```

local.state.Heading = entity->getDyn()->getHeading();
local.state.Speed = entity->getDyn()->getSpeed();
local.state.Health = entity->getStatus()->getHealth();
local._state = true;

local.position.X = entity->pos.x;
local.position.Y = entity->pos.y;
local.position.Z = entity->pos.z;
local._position = true;

local.ident.ID = entity->getId();
local.ident.Type = entity->getStatus()->getType();
local.ident.Color = entity->getColor();
local.ident.IFF = entity->getStatus()->getIFF();
local._ident = true;

```

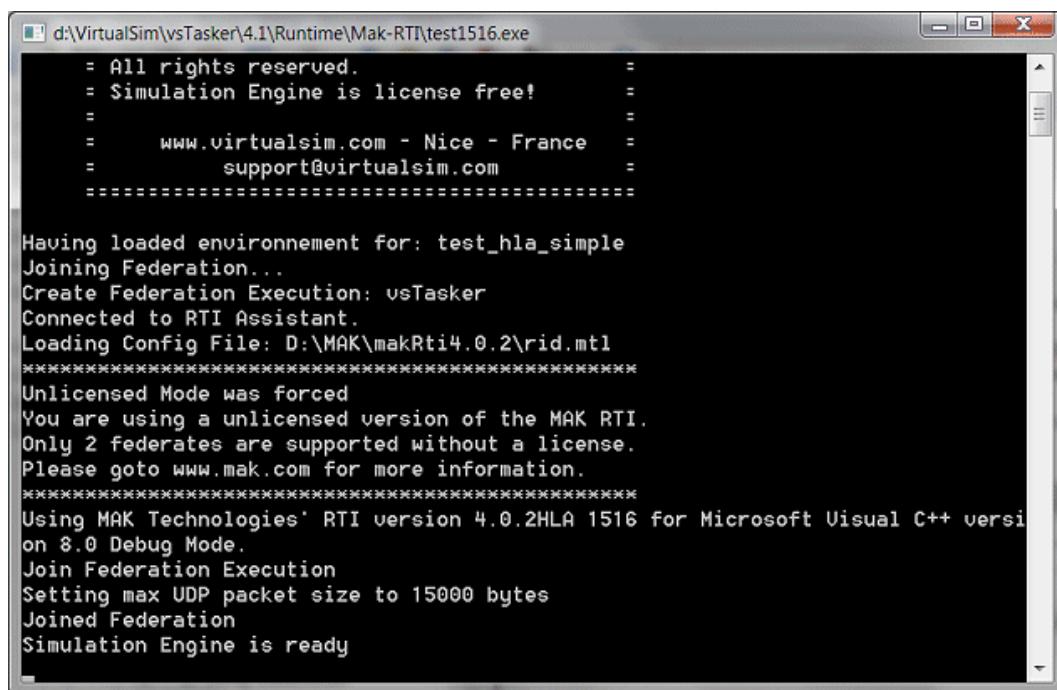


*It is mandatory to use the function `getEntity()` (if the FedItem works with entities) or `getHandle()` (if the FedItem uses objects only) to set the RTI handle before publishing.*

## Publish

Now, you can compile and even run the simulation.

The Simulation Engine will connect to the Mak RTI and register the Federation.



A screenshot of a terminal window titled "d:\VirtualSim\vsTasker\4.1\Runtime\Mak-RTI\test1516.exe". The window displays the following text:

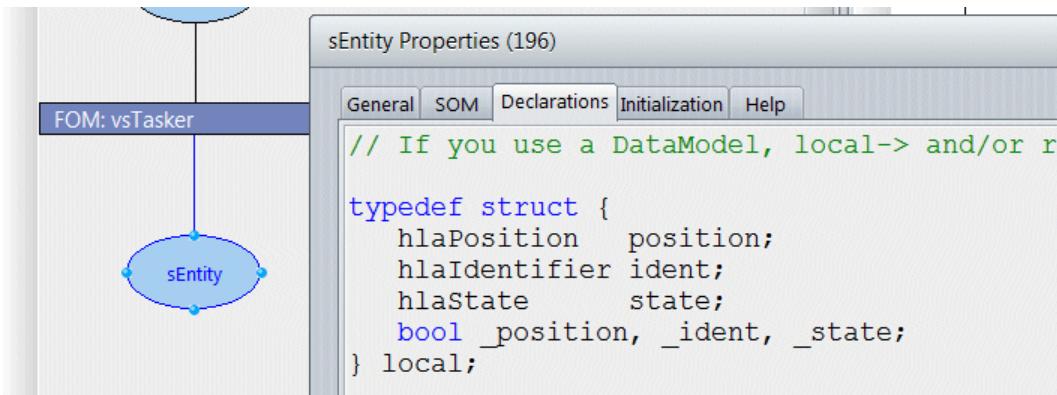
```
= All rights reserved.          =
= Simulation Engine is license free! =
=                                     =
=     www.virtualsim.com - Nice - France =
=           support@virtualsim.com =
=====
Having loaded environnement for: test_hla_simple
Joining Federation...
Create Federation Execution: vsTasker
Connected to RTI Assistant.
Loading Config File: D:\MAK\makRti4.0.2\rid.mtl
=====
Unlicensed Mode was forced
You are using a unlicensed version of the MAK RTI.
Only 2 federates are supported without a license.
Please goto www.mak.com for more information.
=====
Using MAK Technologies' RTI version 4.0.2HLA 1516 for Microsoft Visual C++ versi
on 8.0 Debug Mode.
Join Federation Execution
Setting max UDP packet size to 15000 bytes
Joined Federation
Simulation Engine is ready
```

Let's do the [Subscriber](#)...

# Subscribe

- **Defining the Data**

Now, double-click on the **subscribing** FedItem to call its Property Window



```
typedef struct {
    hlaPosition    position;
    hlaIdentifier ident;
    hlaState      state;
    bool _position, _ident, _state;
} local;
```

- **Getting the Attribute**

Every time an Attribute of a subscribed Object is changed by one federate, the corresponding FedItem is called for the paired Attribute.

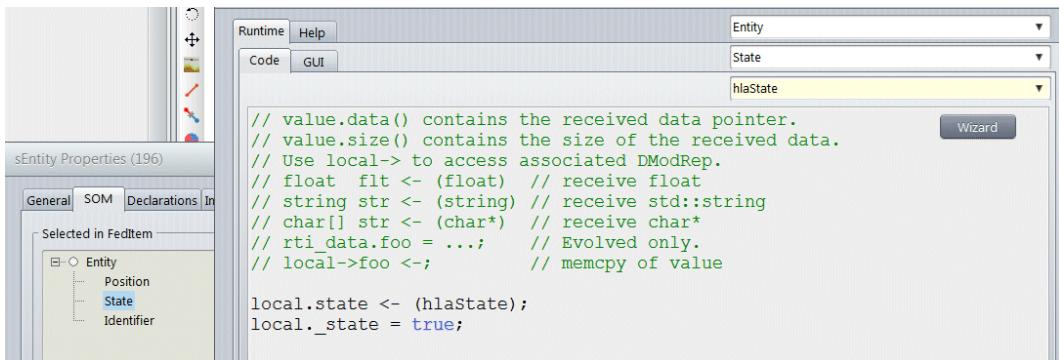
We then need to copy locally the data coming from the RTI.

Let's open the State Attribute and add the following code:

```
local.state <- (hlaState);
local._state = true;
```

This code means that the data coming from the RTI must be casted as `hlaState` (structure) and copied to the `local.state` variable we defined above in the `Declaration` part of the `FedItem`.

## Subscribe



Let's do the same for the remaining two Attributes:

```
local.position <- (hlaPosition);  
local._position = true;  
  
local.ident <- (hlaIdentifier);  
local._ident = true;
```

## • Getting the Object

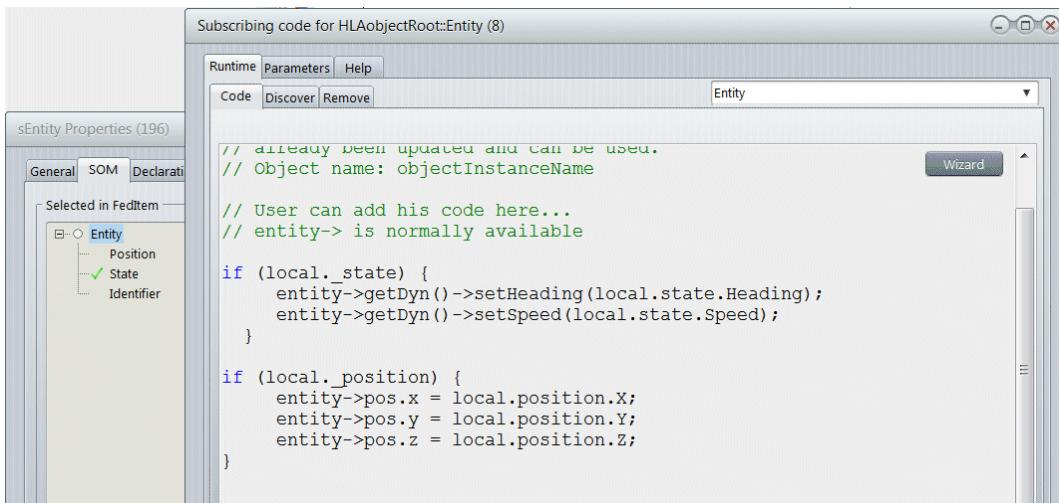
When the Object is first found on the RTI, the Discover part is called with the Object handle.

If the FedItem allows runtime entities ( Allow RT Entities) the Discover part will automatically try to create an Entity, set the entity pointer and pair it with the handle.

```
entity = new Entity("default", objectInstanceName);  
add(entity); // in the used_by list  
store(entity); // associate it with the handle
```

We know that the Object code part will be called once all Attributes have been received.

This is the reason why we have raised the flag, because not all data structure might be changed and as several Entities do share the same FedItem, this way of working insure that data from different entities will not be mixed up.



```
if (local._state) {  
    entity->getDyn()->setHeading(local.state.Heading);  
    entity->getDyn()->setSpeed(local.state.Speed);  
}  
  
if (local._position) {  
    entity->pos.x = local.position.X;  
    entity->pos.y = local.position.Y;  
    entity->pos.z = local.position.Z;  
}
```

Now, let's [run the application...](#)

## **Using DataModel**

This mode must be preferred for 1516 and large Federations.  
The Wizard can then be used and will reduce the coding effort.

This mode relies on data structures generation extracted from the Federation definition.

It also can work with user defined data structures if these one matches their RTI counterpart.

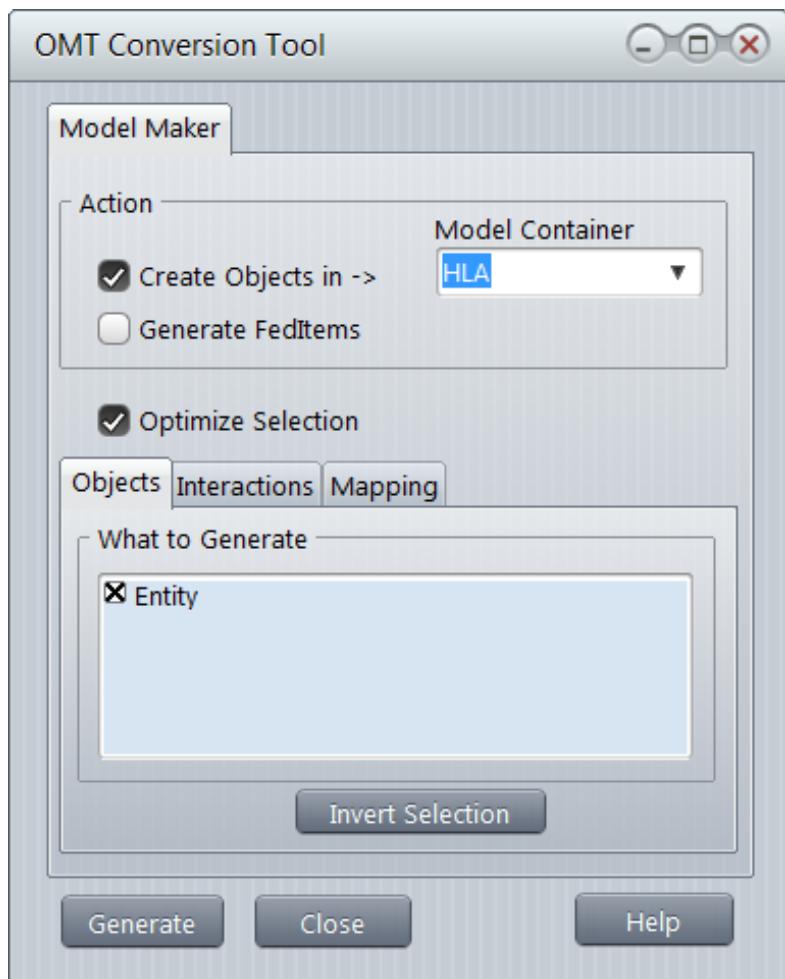
### **• Generating Data Models**

With 1516 Federations, vsTASKER produces the OMT like objects that correspond to the Federation definitions.

It is then possible to make C++ data structures based on these definitions, in order to directly use them into the code.

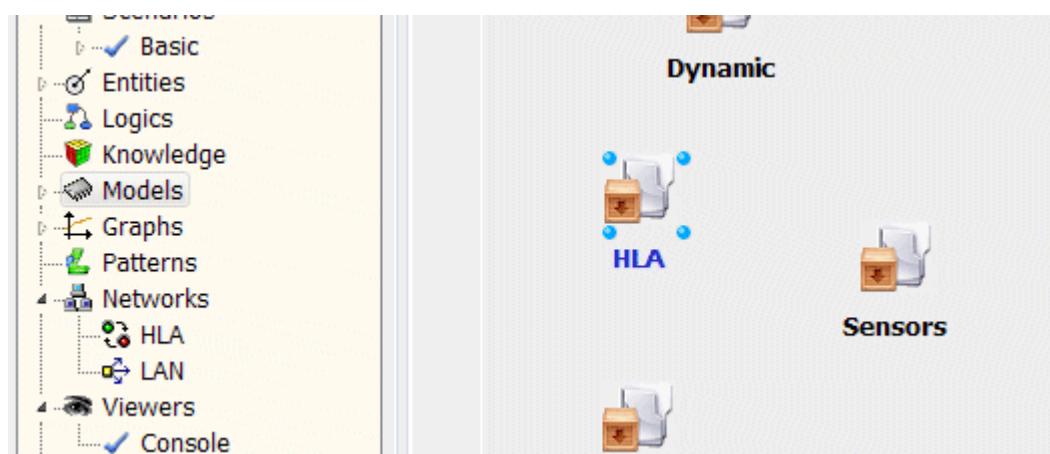
Attaching a DataModel to a FedItem will also speed up the data conversion and manipulation from RTI to Entity data structures.

To make (or even update) the DataModel based on the OMT definition, go to OMT panel then right click and select **OMT Converter Tool...**



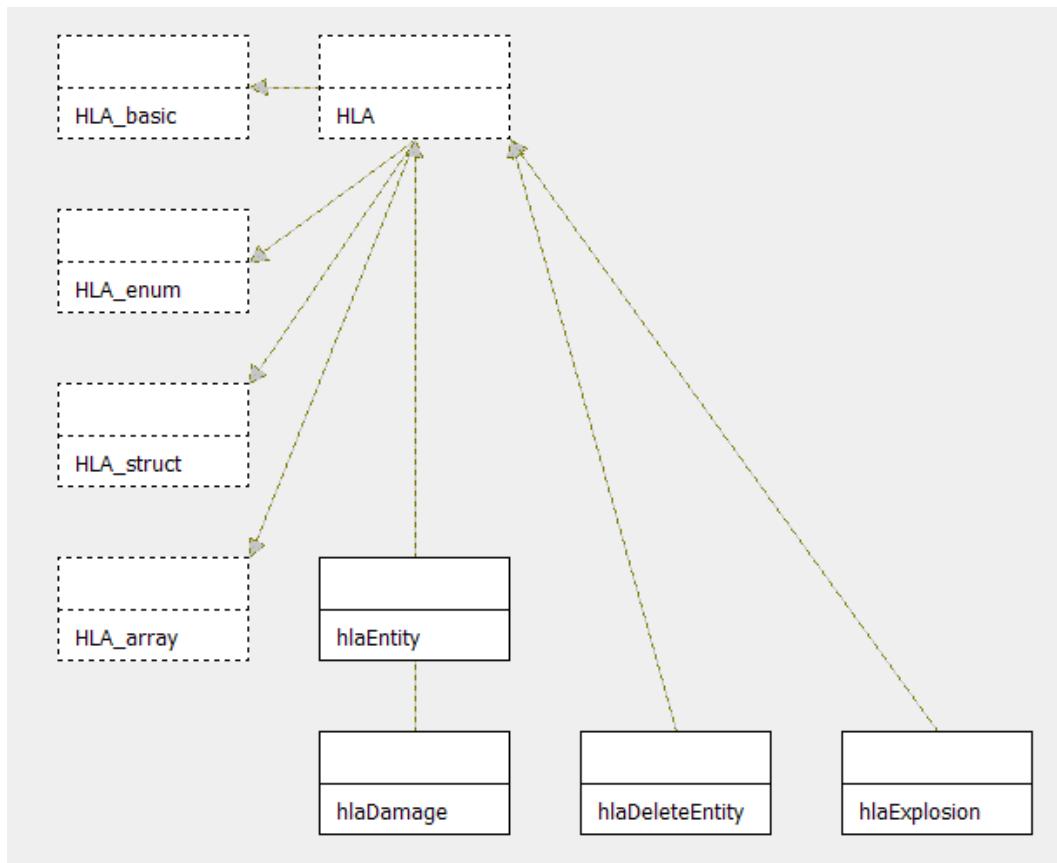
Using this converter tool, you will be able to select all Objects and Interactions defined into the FOM and make them DataModel objects into Container named HLA (by default. This can be changed).

Press **Generate** then, open the container **HLA** into Models



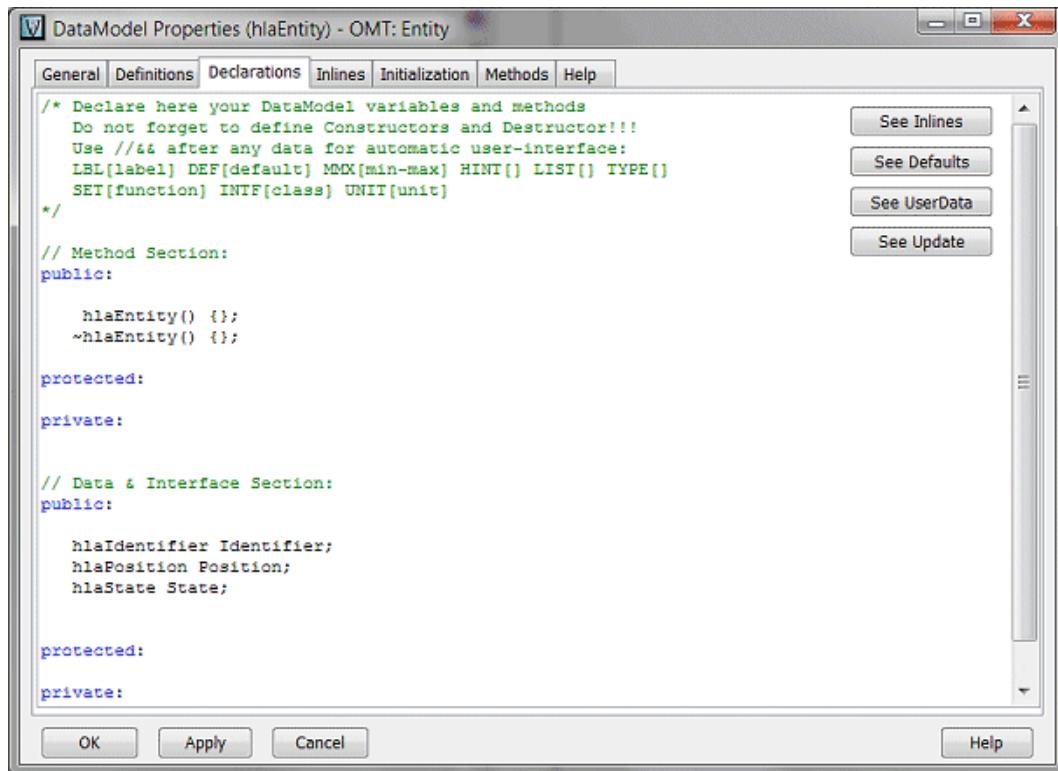
## Using DataModel

We can then see that DataModels have been generated:

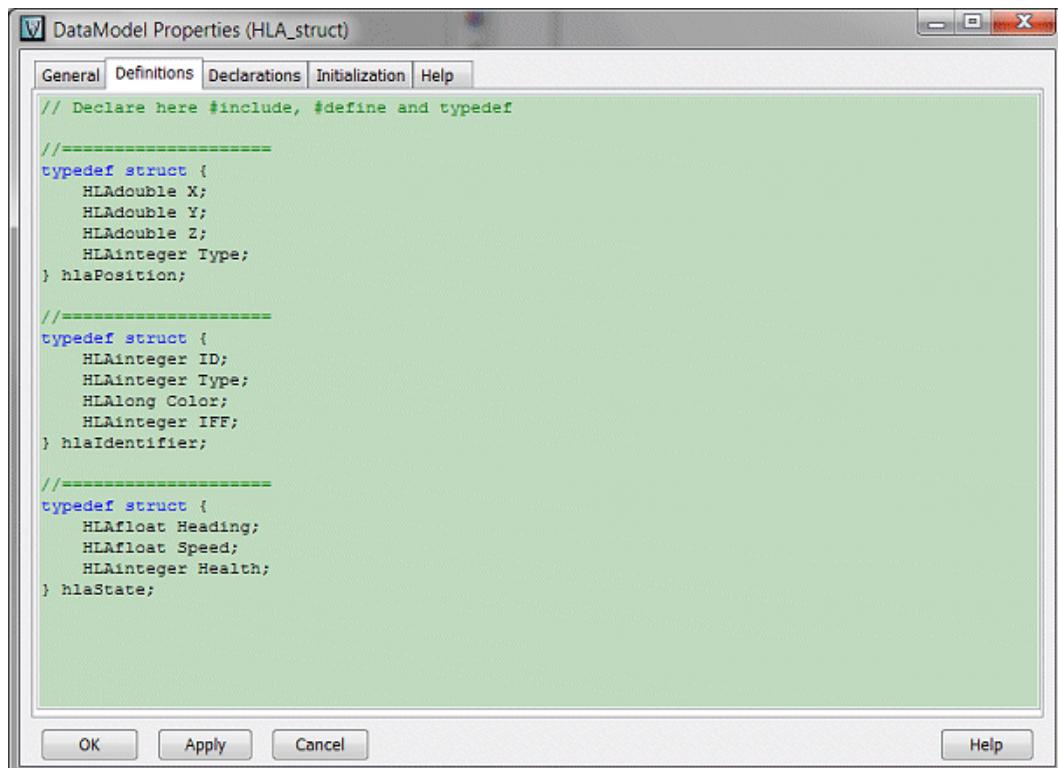


Here, we have all Federation Objects and Interactions ([hlaEntity](#), [hlaDamage](#), [hlaDeleteEntity](#), [hlaExplosion](#)) that have been defined as DataModel classes. For example, if we open [hlaEntity](#) DataModel, we will see the following:

## Using DataModel

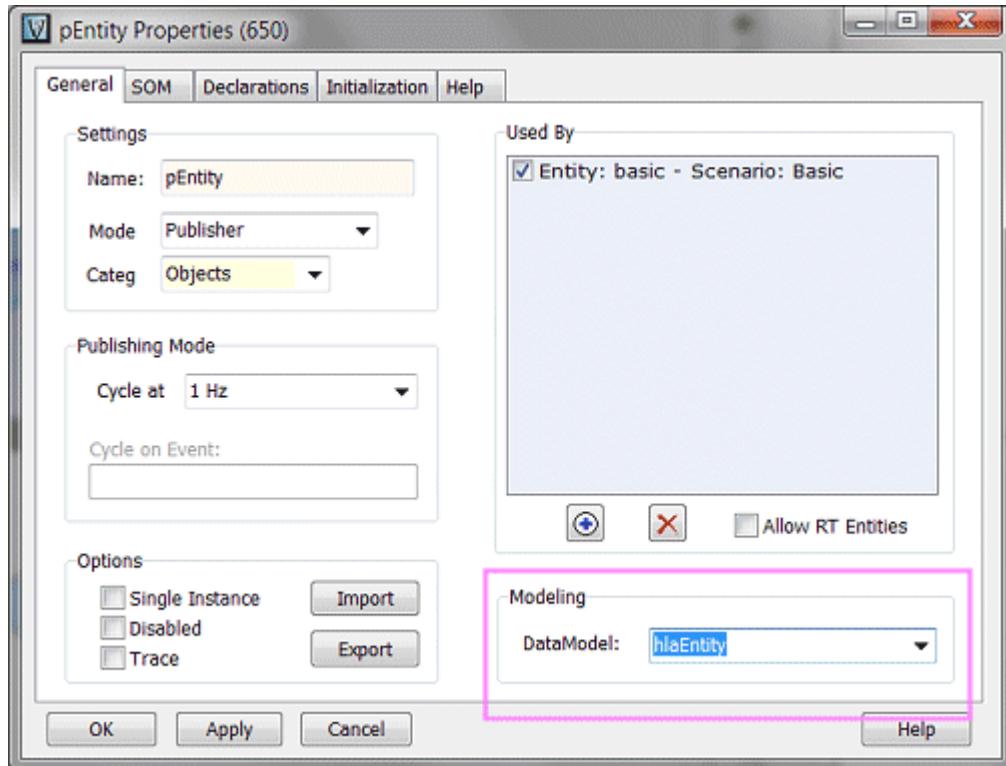


So the class [hlaEntity](#), contains the three Attributes [Identifier](#), [Position](#) and [State](#), whose types has been defined in DataModel [Hla\\_struct](#):



## Using DataModel

So now, it is possible to attach each DataModel object to their corresponding FedItem, allowing then vsTASKER to generate code that will simplify the data transfer.



Time to define the SOM data transfer for both Publisher and Subscriber FedItems.

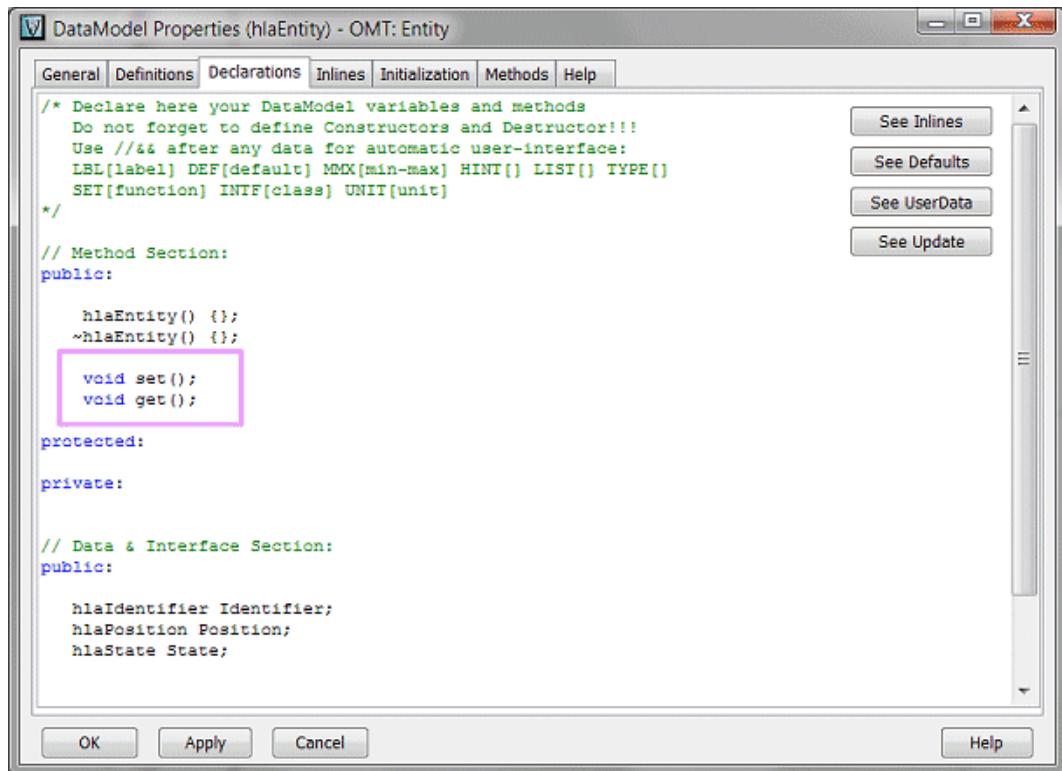
### • Setting/Getting the Data

The data models built by the OMT converter are just class definitions. The user must map them with real simulation data before using them.

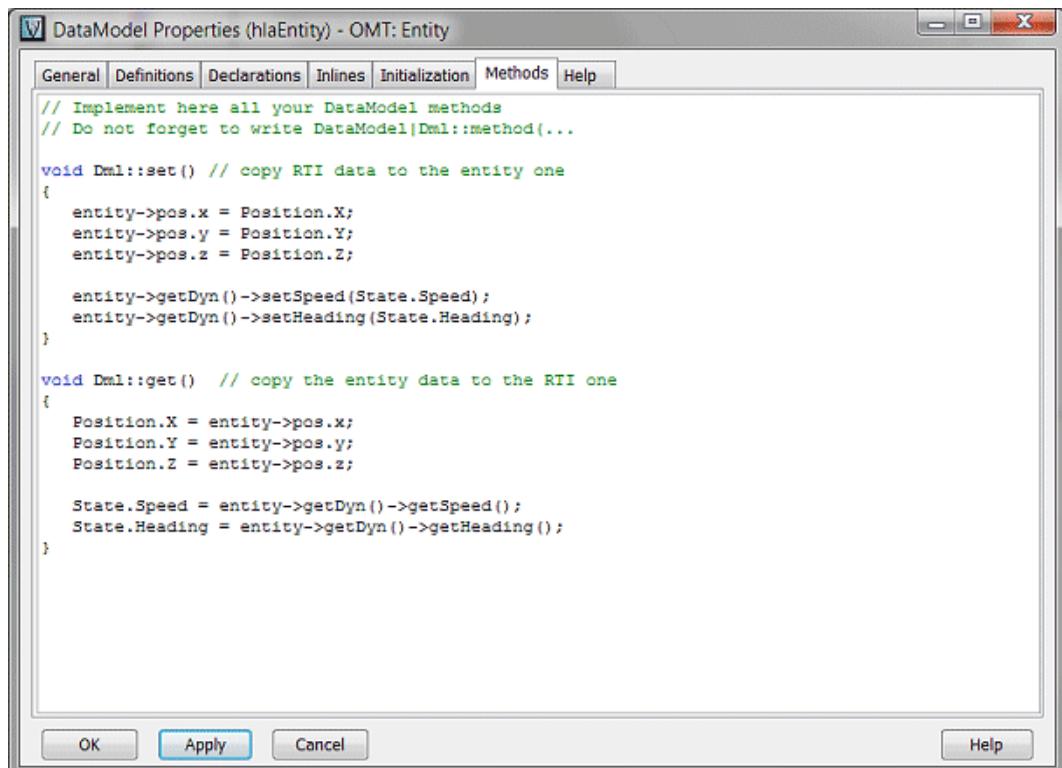
This is not mandatory. For example, the Federation can contain some Objects and Interactions that are specific to an environment and have no connection with vsTASKER internal data (belonging to Entity Scenario or whatever items).

But in our case, hlaEntity object contains position and state values that must be copied to/from existing runtime values, belonging to the entity. To do that, we will create two methods `set()` and `get()`.

## Using DataModel



and we will define each of them:



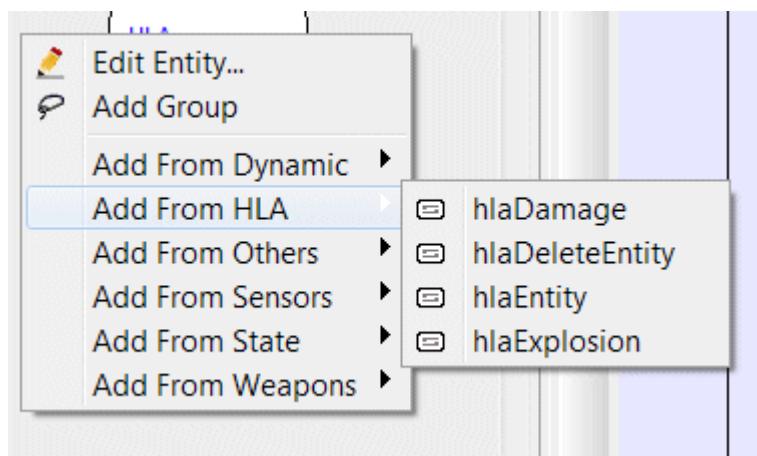
## • Giving DataModels to Entity

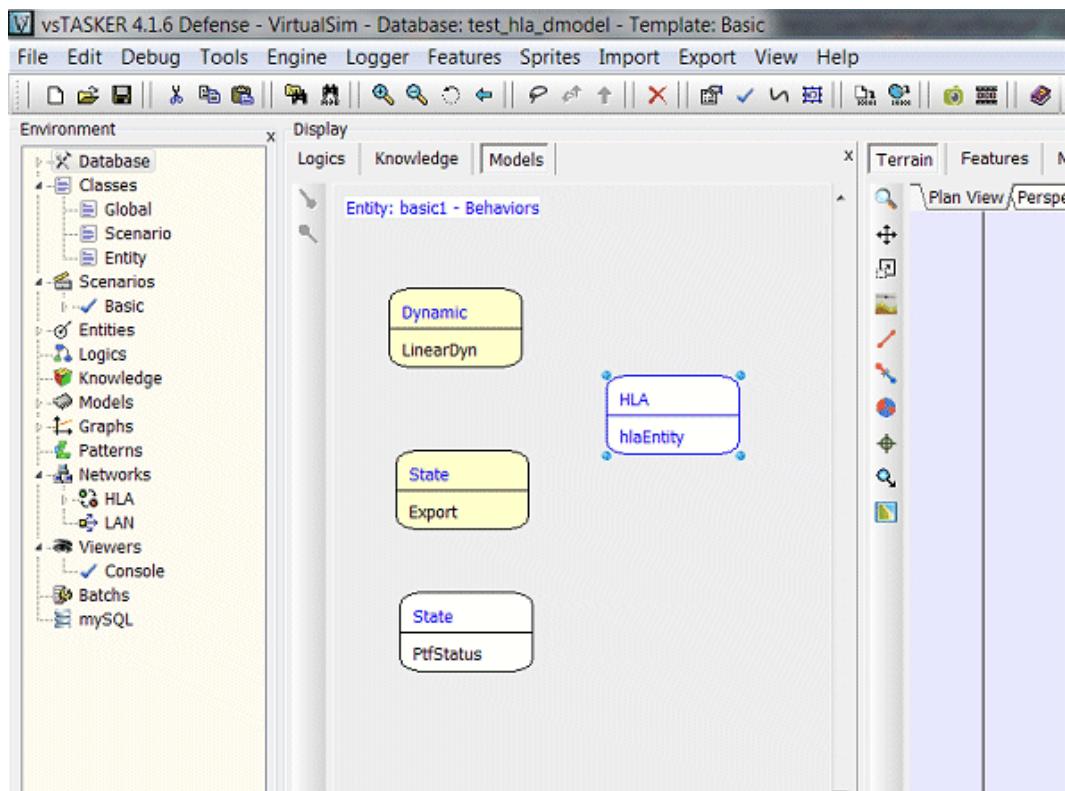
We want each Entity using the FedItem to hold one [hlaEntity](#) DataModel.

For that, we will attach the DataModel to the Entity of the scenario, **but also to the default one of the Catalogs!**

This is because the subscribing FedItem will create external entities based on the [default](#) Entity (in this case).

You add the DataModel to the Catalog Entity the same way as for a Scenario Entity, by selecting it and switching the [Models](#) panel.





## • Understanding things

Entity has data distributed between several internal structures, data-models and components.

This data does not necessarily match the one requested for HLA input/output of Objects/Interactions.

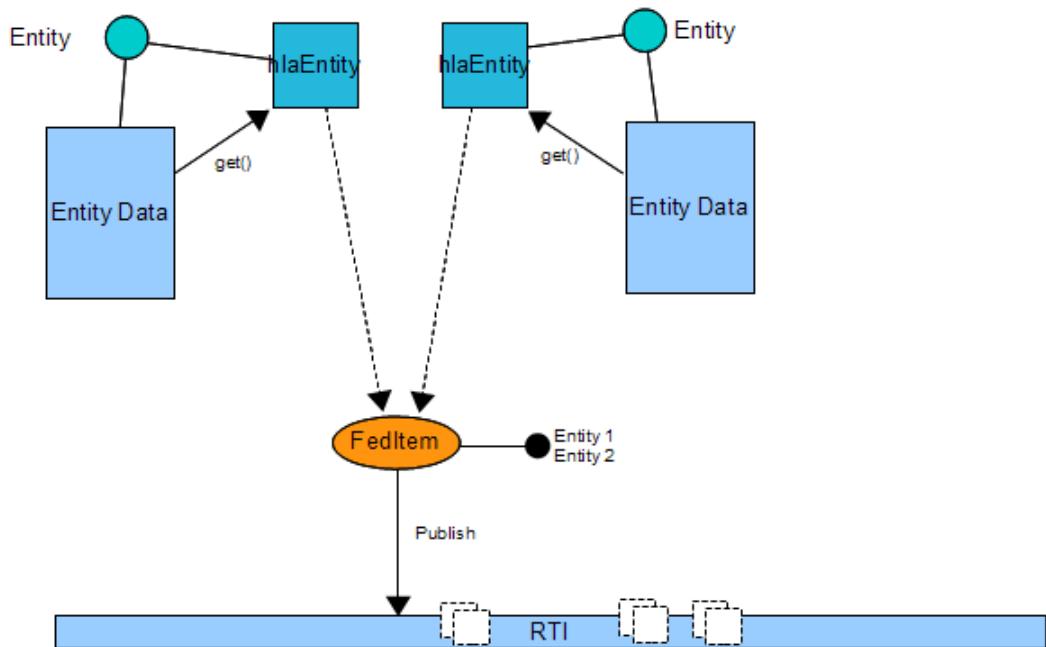
This is the reason why we need to attach to the Entity every Data-Model generated by the OMT converter.

According to the following picture, we can see that one FedItem is instanciated, even if used by Entity 1 and 2.

As a publisher, the FedItem will be called by the RTC at the frequency set by the user, then, will process the Object code for each entity attached to it (used\_by list), then, call the code of all Attributes.

We will see later how to get a pointer to the `hlaEntity` data of the entity. Once obtained, using the `get()` function will refresh the data. Then, FedItem will publish it.

## Using DataModel



The subscribe part is the reverse process of the publish.

There is also one FedItem per HLA Object subscribed even if several entities are using it.

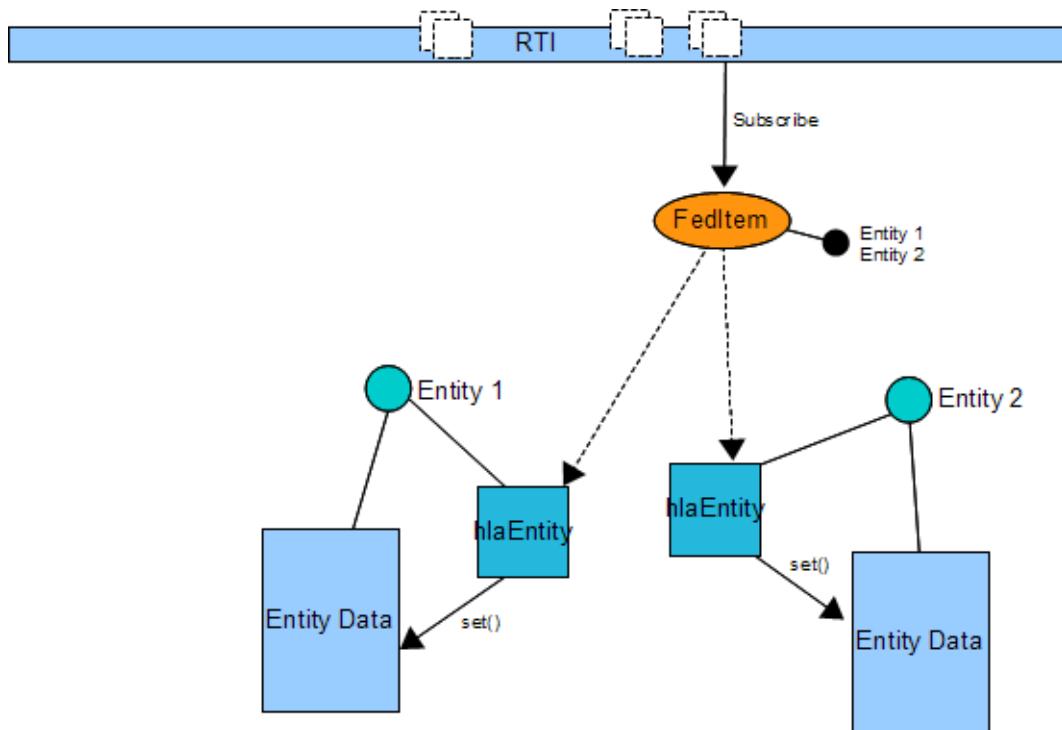
We the subscriber FedItem receives a data from the RTI, it will retrieve the corresponding entity automatically and then, get the pointer to the `hlaPosition` of the entity.

This is done because the FedItem is instructed to use `hlaEntity` as a DataModel, assuming that every attached entity holds the `hlaEntity` DataModel also.

If not (forgotten by the user) nothing will really happen.

Once all Attributes have been received from the RTI, the Object code is then called. There is when the `set()` function is called to refresh the entity data values.

## Using DataModel



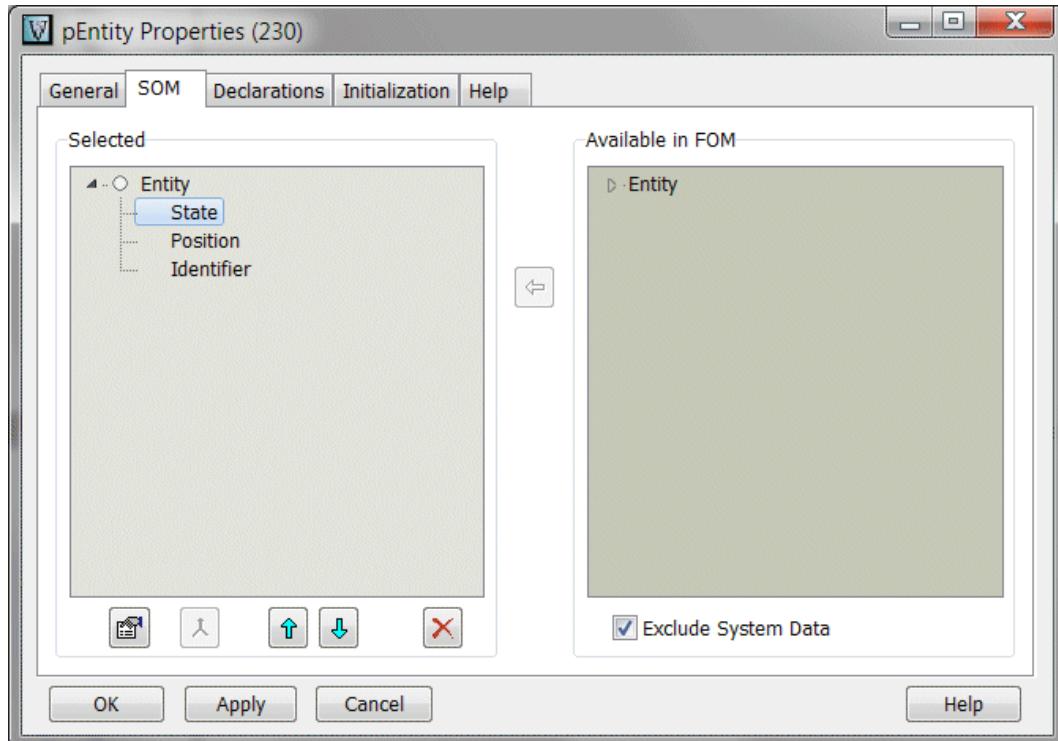
Now, let's define the [FedItems](#).

## Publish

# Publish

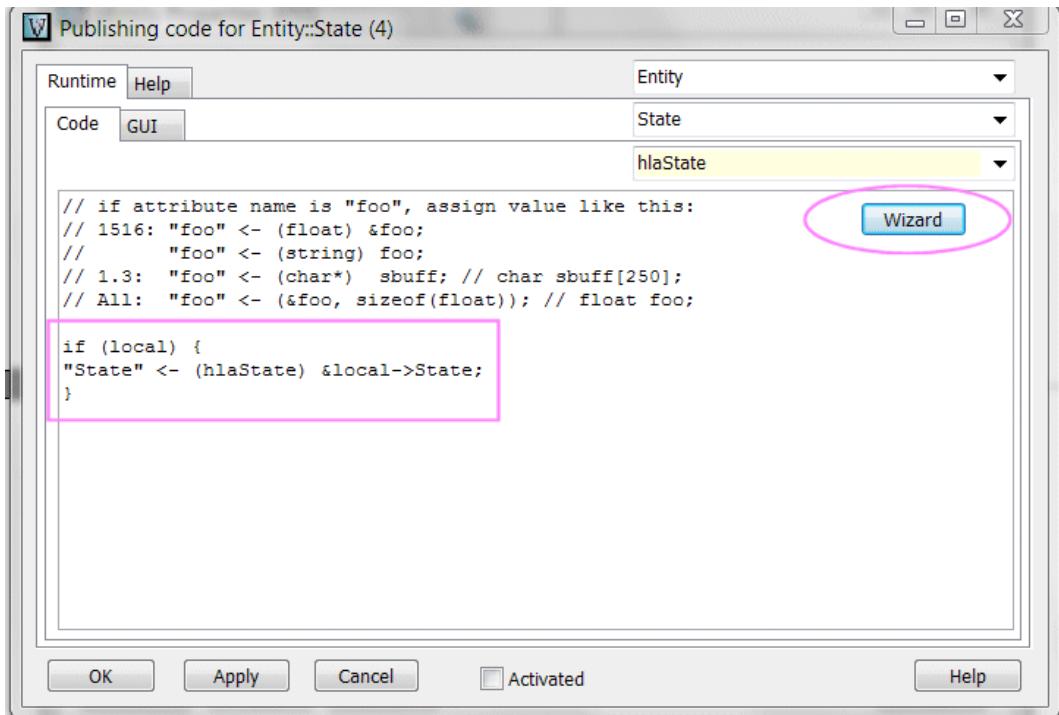
- **Defining the Data**

Now, double-click on the **publishing** FedItem to call its Property Window



Here, we can see that Entity Object has been selected and that the three Attributes ([State](#), [Position](#) and [Identifier](#)) are listed.

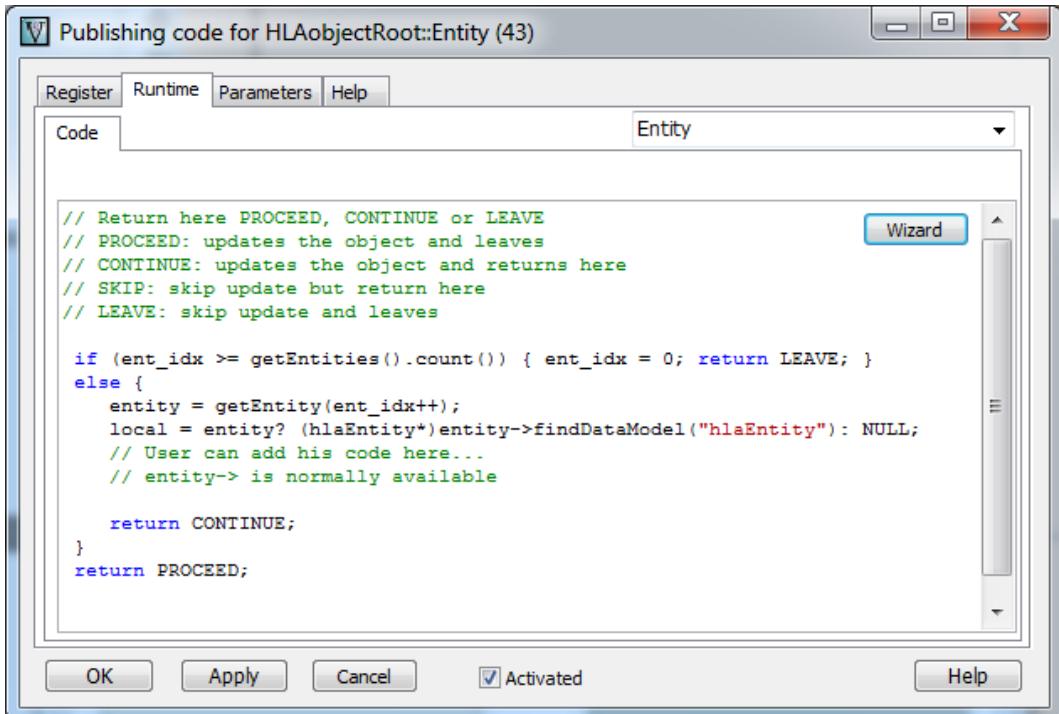
Let's open the [State](#) one and click on [Wizard](#) button to automatically get the correct code.



Do it now for all Attributes ([Position](#) and [Identifier](#))

For the [Entity](#) Object, we just need to set the [local](#) data pointer by retrieving the [DataModel](#) attached to the entity.

This is done just after having set the [entity](#) pointer:



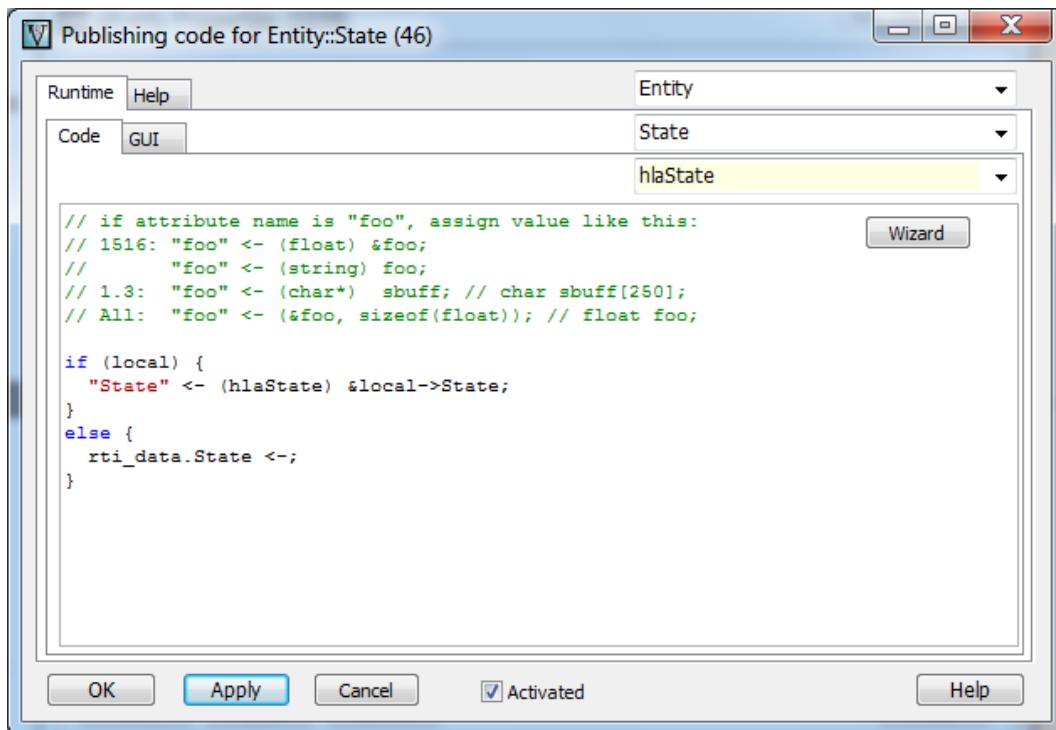
## **Publish**

```
local = entity? (DataModel*)entity->findDataModel("DataModel"):  
NULL;
```

# Subscribe

- **Getting the Attribute**

Now, double-click on the **subscribing** FedItem to call its Property Window then, for each Attribute, use the Wizard button to get the automatic code. You must have set the DataModel to [hlaEntity](#) in [General::Modeling](#)



Here, we can see that if the `local` pointer has been correctly set to the entity [hlaEntity](#) object, then `local->State` will receive the data coming from the RTI, otherwise, `rti_data` will receive it (and the user will need to transfer it from the Object code).

*vsTASKER generates a union of pointer names: local and dmodel. You can use either one.*

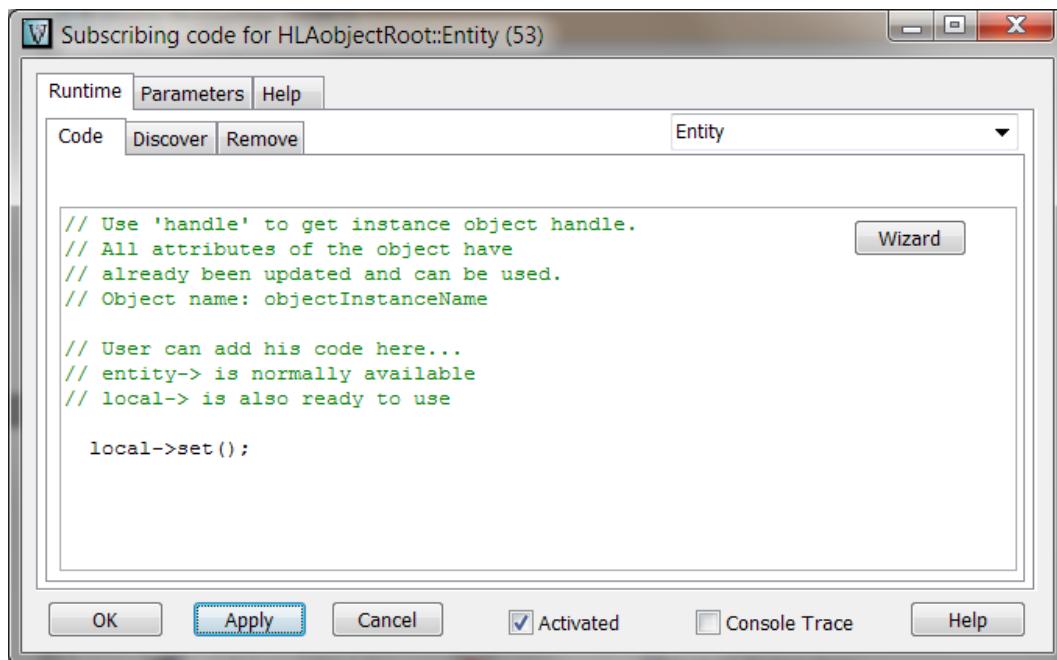


*Wizard after version 6.0 uses dmodel instead of local, as dmodel refers to the attached data\_model.*

Now, let's open the Entity object of the FedItem.  
This part will be called when all Attributes have been updated.

It is then time to copy the data from [hlaEntity](#) to internal entity structure, using the `set()` function defined in the [hlaEntity](#) DataModel.

## Subscribe



Now, we can compile and run the simulation.

*The entity pointer is normally set into the Discover panel if the FedItem is sensitive to runtime entities ( Allow RT Entities), and is associated with the Object handle.*

*Local pointer is normally set during the discover stage by retrieving the DataModel from the entity created: local = entity? entity->findDataModel("DataModel") : NULL;*



# Running the Sample

- **Setting the second vsTASKER**

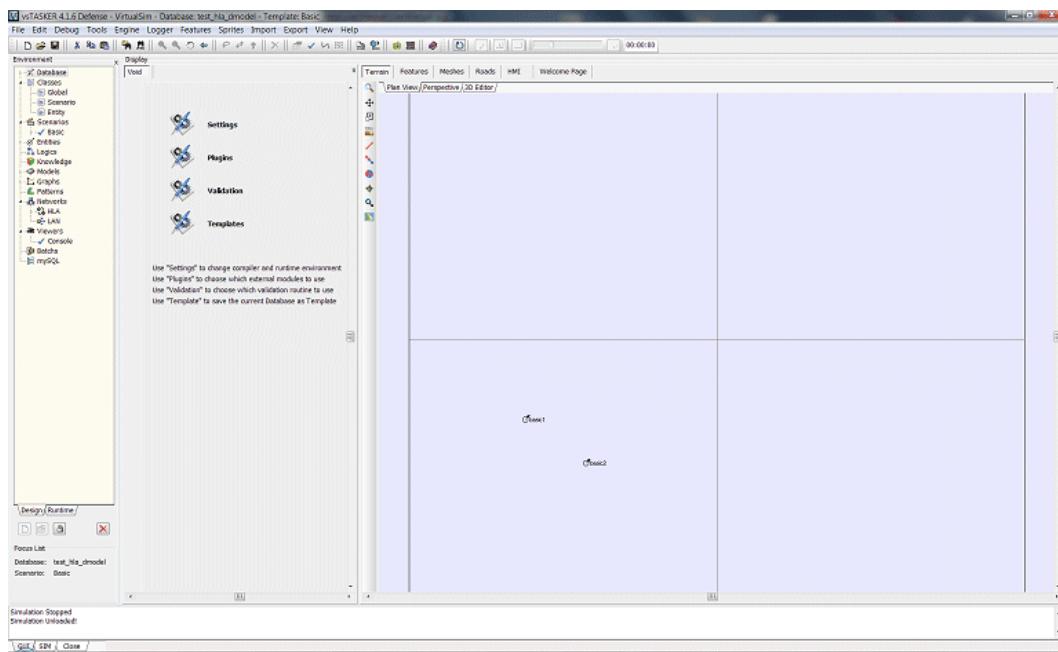
In order to distribute a simulation, we need to have two Federates. In our case, we will just duplicate the current database.

You can duplicate the basic Entity (basic1, basic2) but then, do not forget to add all entities to the Used-by list of the publishing FedItem.

So, just [Save-As](#) the current database with name hla\_sample2

You can also change the background color of the map to distinguish the two databases.

Generate the code, compile and load the simulation



Now, to open the second vsTASKER onto the same computer, you need to execute the batch file **vstasker2.bat** located into base directory.

This batch file must be executed under **Administrator** mode.

The simplest way to do it is to open a **Console** (cmd) under Administrator rights using the [Accessory](#) folder of Windows **Start::Programs**. Then, `c::; cd %vstasker_dir%; then vstasker2`

A new vsTASKER gui will appear. This one will create another shared-memory segment so, it will not conflict with the first one.

## Running the Sample

Load the [hla\\_sample](#) database (the [hla\\_sample2](#) is currently loaded by the first vsTASKER, from the [Save-As](#) command).

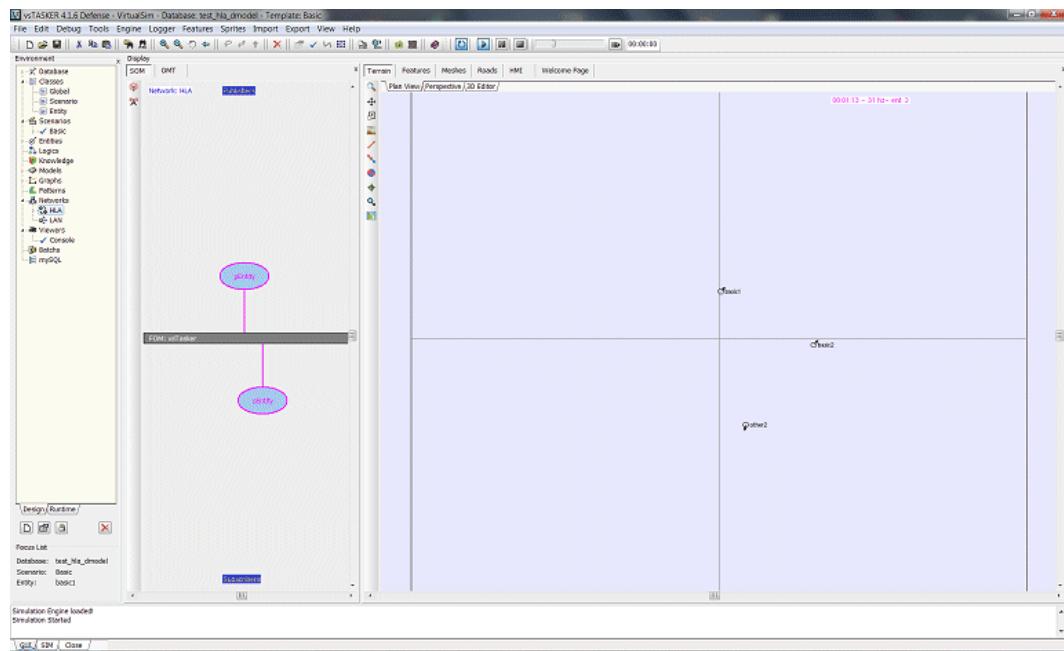
You can then change also the background color.

Rename the Entity and move it somewhere else (to avoid overlaps).

You can also duplicate the current one.

Now, load all simulation and start them.

You will see on both scenario map the local entity and the others sharing the same area.



# My Own Federation

With vsTASKER, it is also possible to define a Federation from scratch and use it quite immediately without too much coding.

The OMT builder and the code generation mechanism will simplify a lot the process.

Let's say for the sake of this example that we want to create one Federation that will contain one Object `myEntity` and one Interaction `myCreate`.

The Object `myEntity` will contain two Attributes: `X` and `Y`, both integers.

The Interaction `myCreate` will contain two Parameters: `posX` and `posY`, both floats.

`myEntity` will be used to update the position of any runtime created entity in either vsTASKER (blue or red).

Whenever the user will create a new Entity in one vsTASKER map, it will appear automatically in the other one.

`myCreate` will be used to react at the mouse click on the map during runtime.

The mouse position (converted into the terrain coordinates) will be sent to other vsTASKER.

A red circle will be displayed at the position for 3 seconds.

Now, we need to create the blue Database.

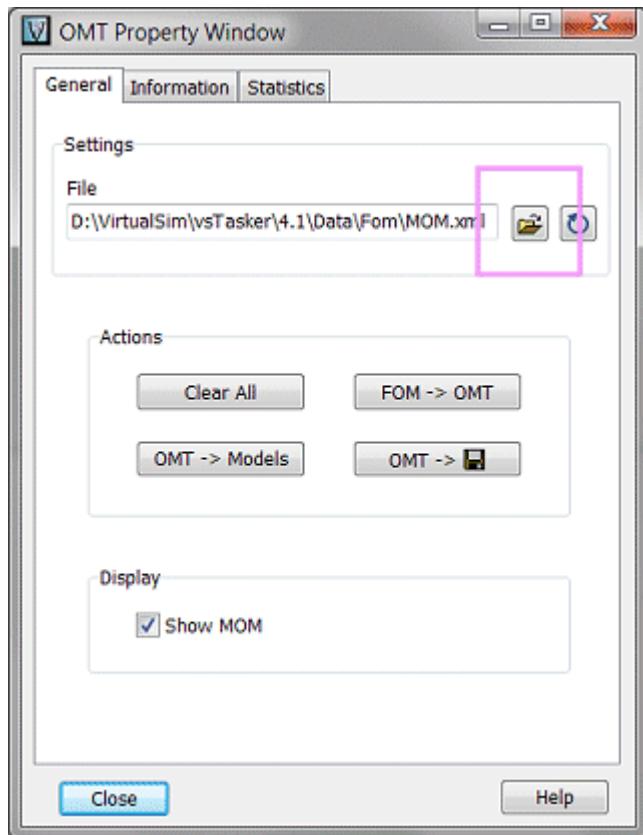
Let's select the Basic template for all.

## Defining the FOM

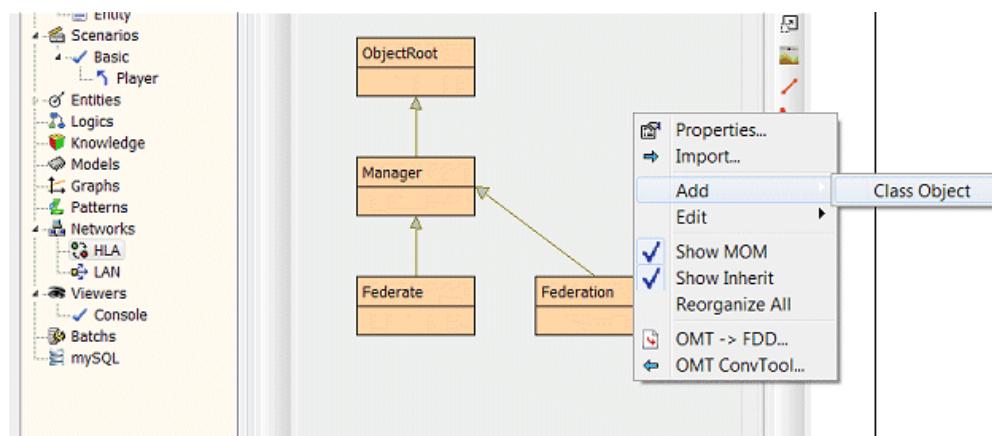
# Defining the FOM

- **myEntity Object**

Go to the HLA::OMT panel, then select Properties and load a **MOM** file into **Data/Fom** directory (change the file type from OMT to XML)



Now, let's define the Object **myEntity**:



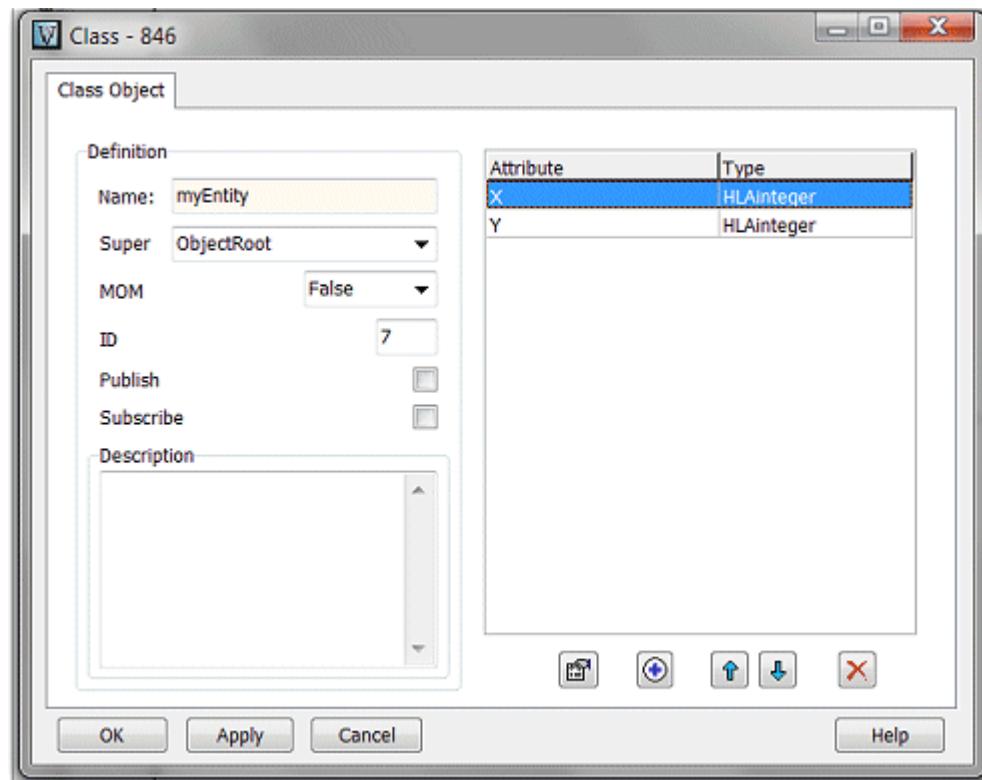
## Defining the FOM

Double-click the object to get to the Property window.

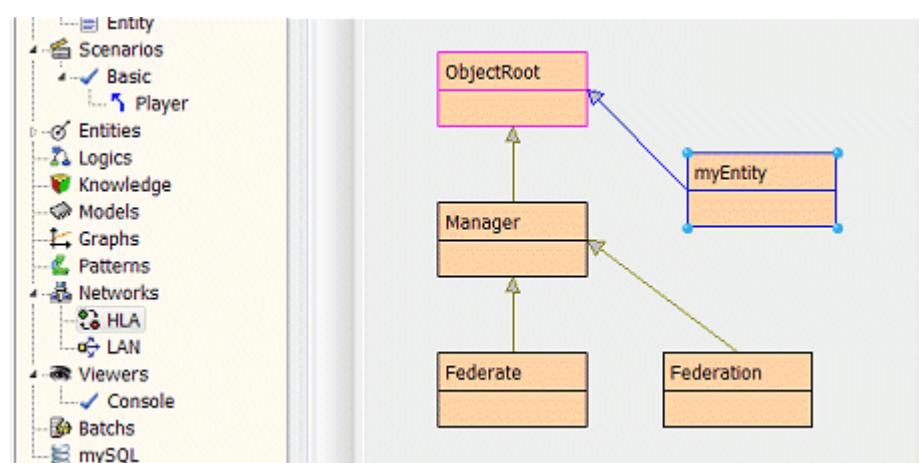
Set the Name to myEntity.

Set the Super to ObjectRoot

Add the two Attributes by defining each of them: (Name: x, Type: HLAinteger),  
(Name: y, Type: HLAinteger)



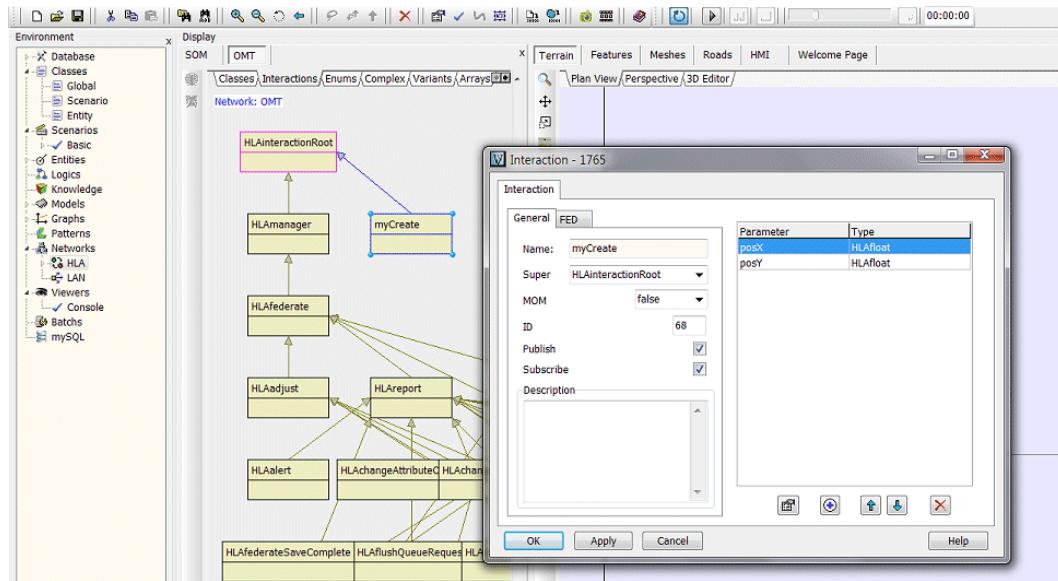
Now, we have defined the object myEntity:



## • myCreate Interaction

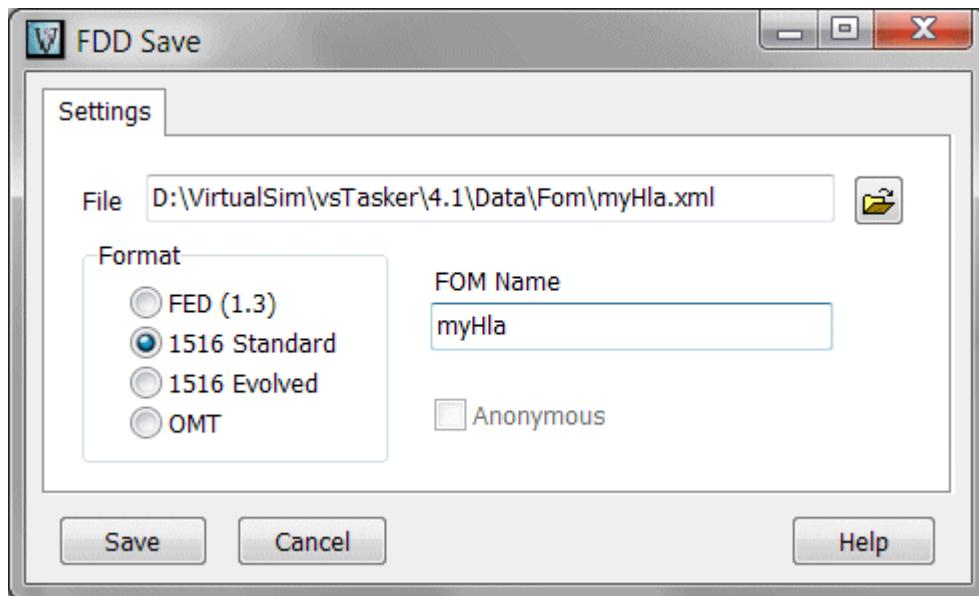
## Defining the FOM

Let's do the same things for Interactions panel.



## • Making the FOM

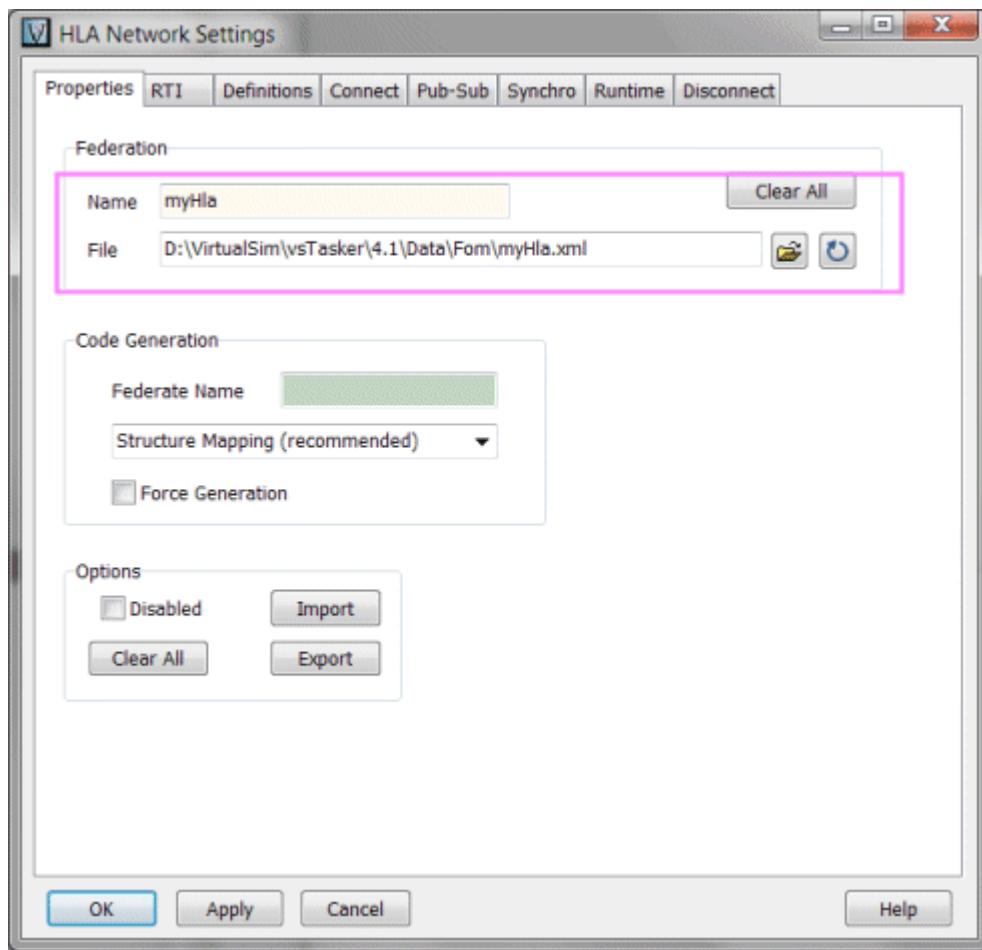
Now, right click the Diagram and select OMT -> FDD



We will select 1516 FOM, give a name to the Federation and store it as an XML file in Data/Fom directory.

Now, we will load it (this is not automatic)

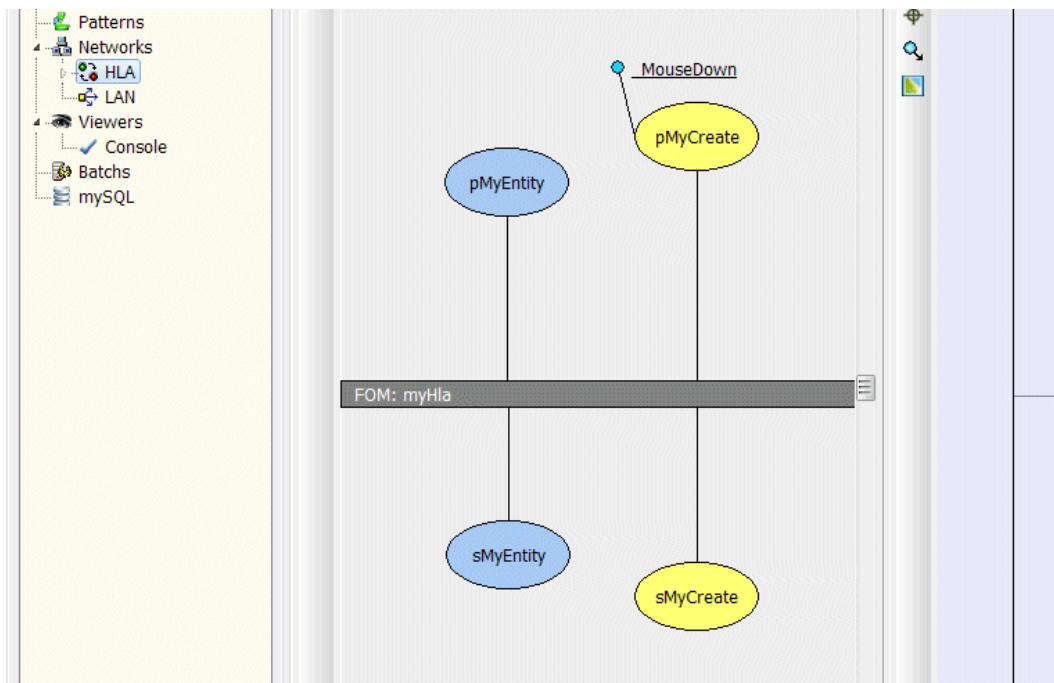
## Defining the FOM



Now, it is time to define the [FedItems](#)

## Defining the FedItems

# Defining the FedItems



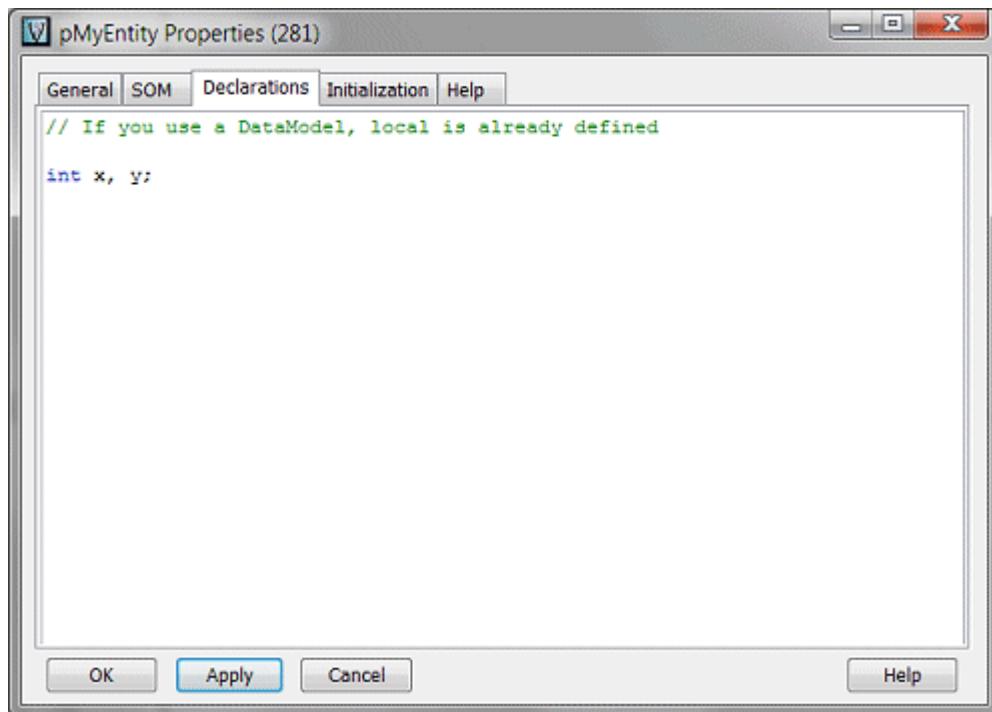
# Publish

- **Publisher Object**

Because the Federation Object is quite simple, we will just use the **Direct Access** method. We keep the **DataModel** one for complex Federations.

Let's start with the Object **myEntity**:

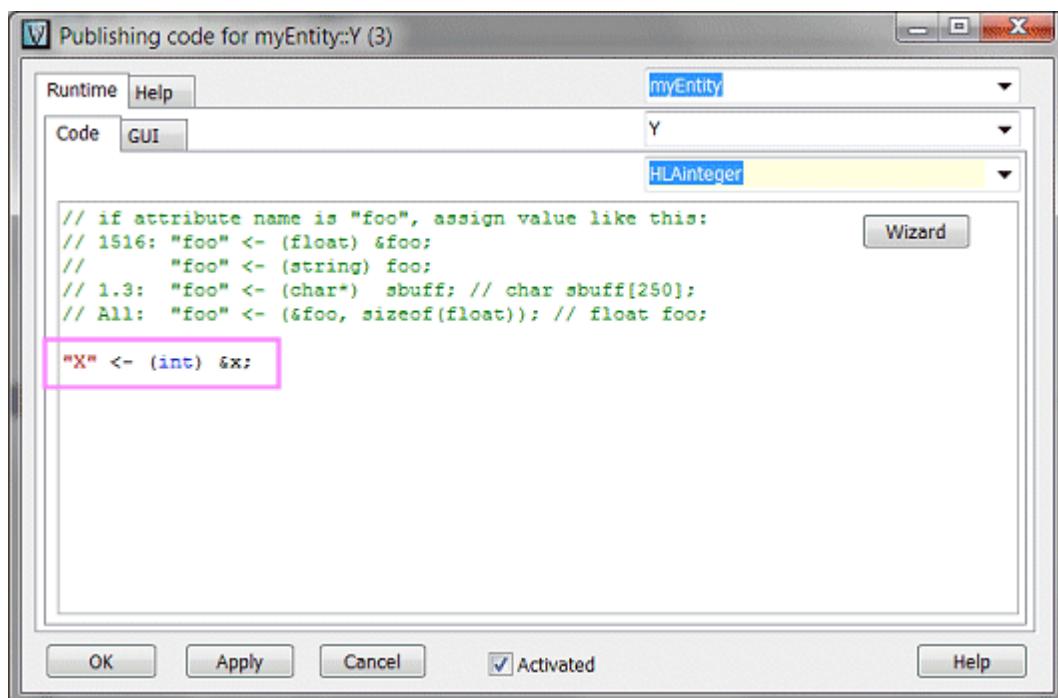
We start by defining locally the data that will hold the values to be sent to the RTI.



Here: `int x, y;` because we know that **myEntity** Object got two Attributes **X** and **Y** of **integer** type.

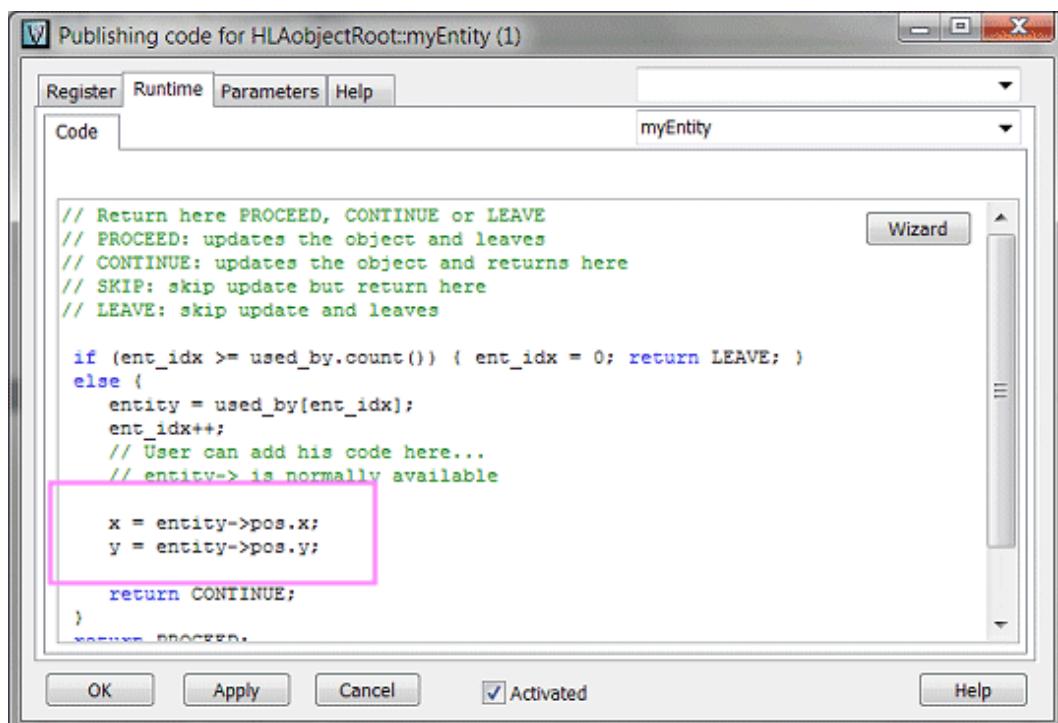
Now, for each Attribute, let's do the copying:

## Publish



```
"X" <- (int) &x;  
"Y" <- (int) &y;
```

As we are on the **publisher** object, the base code is called before the Attribute one. So, it is at **myEntity** Object level that we will do the setting of **x** and **y** variable:

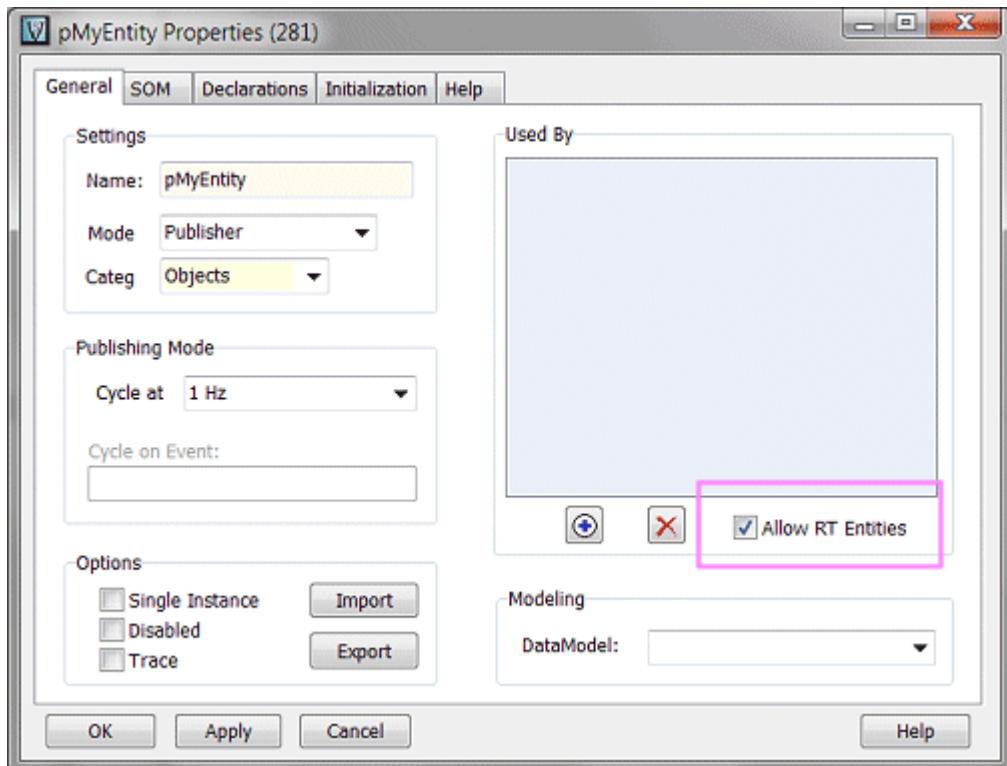


```
x = entity->pos.x;
y = entity->pos.y;
```

So, now, we want the FedItem to work with every entity the user will create at runtime.

Each FedItem can work with a list of entities from the design scenario.

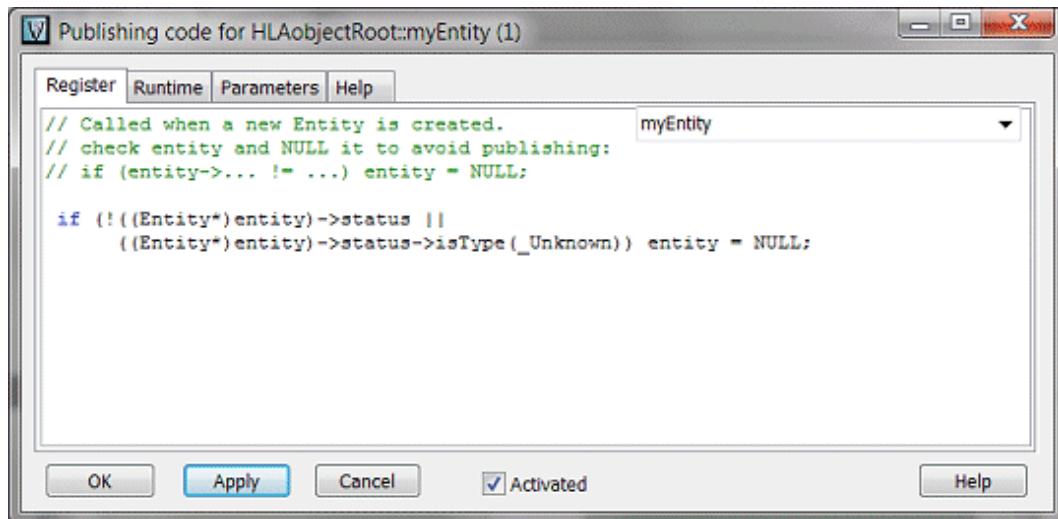
But if we want the new one to be automatically added into the list, we need to select the **Allow RT Entities** field:



This lead us to the filtering of new runtime entities, because maybe, not all runtime entities have to be using this FedItem.

So, the user can add a filtering code here:

## Publish



```
if (!((Entity*)entity)->status ||  
    ((Entity*)entity)->status->isType(_Unknown)) entity = NULL;
```

Here, if the new runtime entity has no Status or is of Unknown type, then, it will not be added to the FedItem.

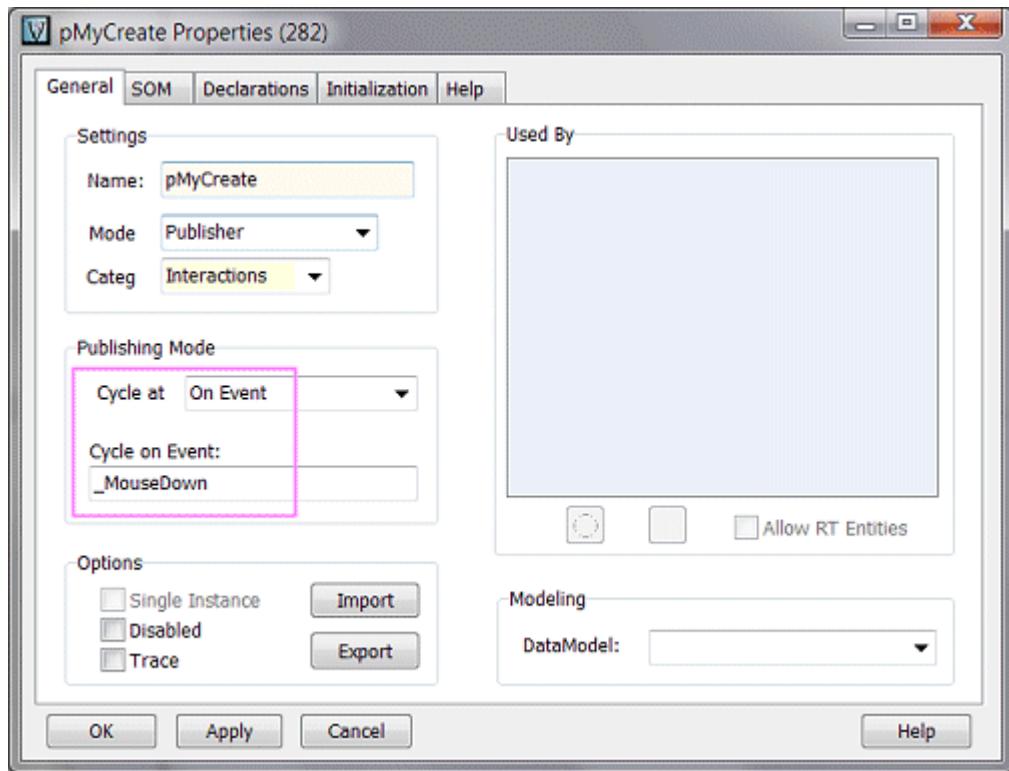
## • Publisher Interaction

Now, time to add the Interaction [myReact](#).

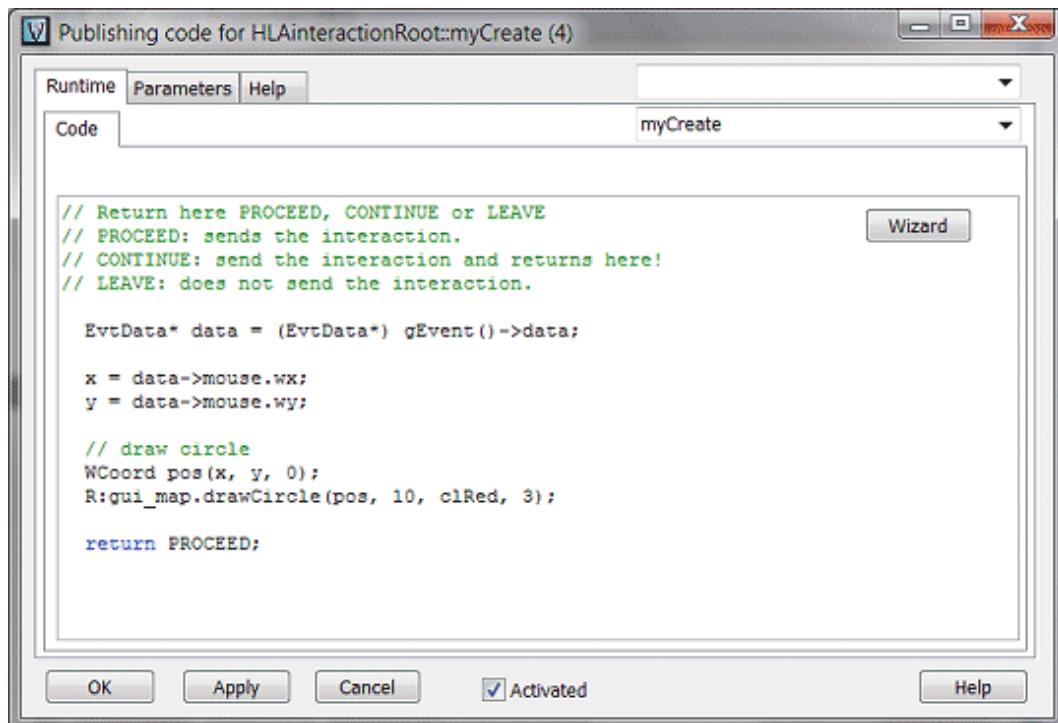
This interaction will be published at every mouse click, producing at runtime the event `_MouseDown`.

For that reason, we will give the Interaction an *On Event* triggering mode.

Let's activate the FedItem with this event:



Now, in the `myCreate` Interaction code, let's extract the world coordinate position of the Mouse:



## Publish

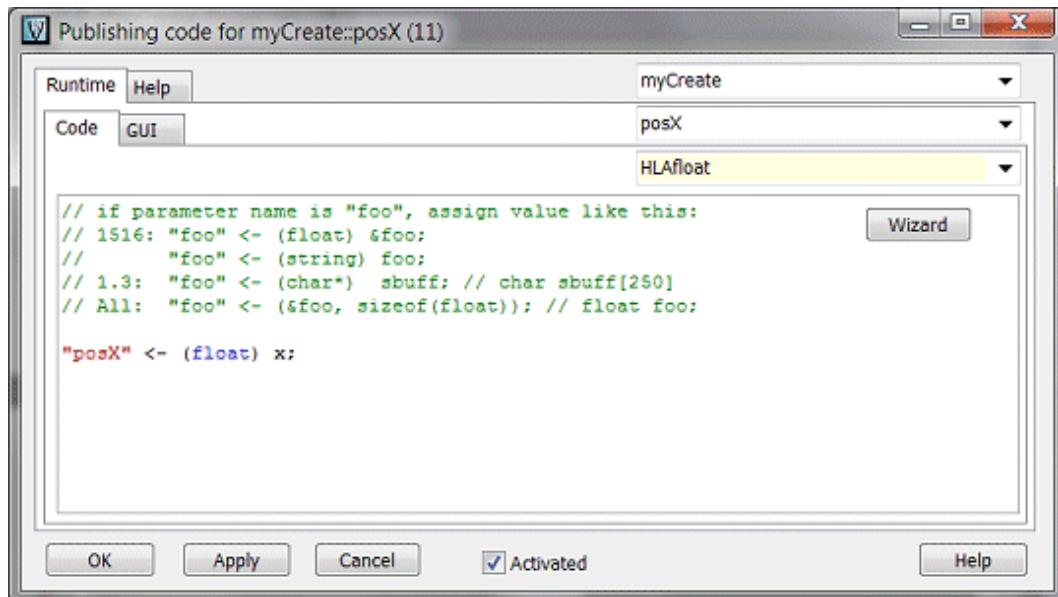
```
// the event carries runtime data (see runtime.h)
EvtData* data = (EvtData*) gEvent()->data;

// we need the mouse.w world coordinates.
x = data->mouse.wx;
y = data->mouse.wy;

// now, we draw a circle, color red for 3 sec
WCoord pos(x, y, 0);
R:gui_map.drawCircle(pos, 10, clRed, 3);

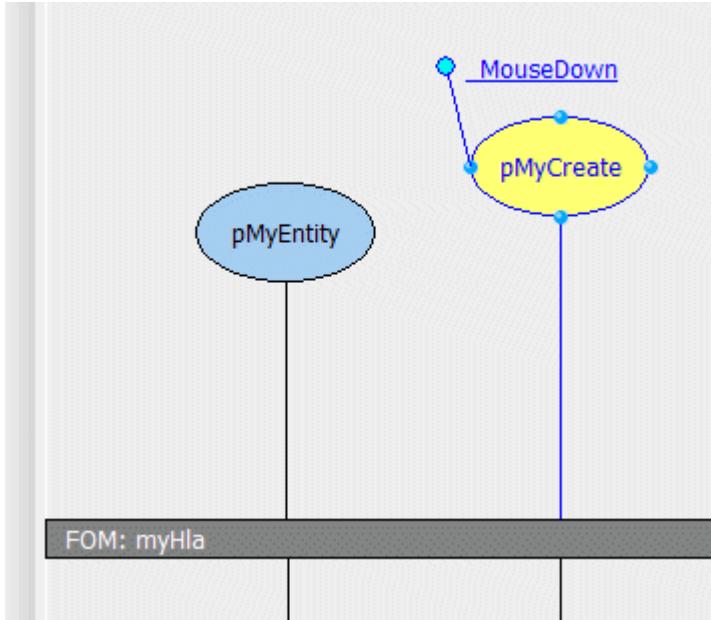
return PROCEED;
```

Time now to set the **posX** and **posY** parameters:



And that's it for the publishing side:

**Publish**



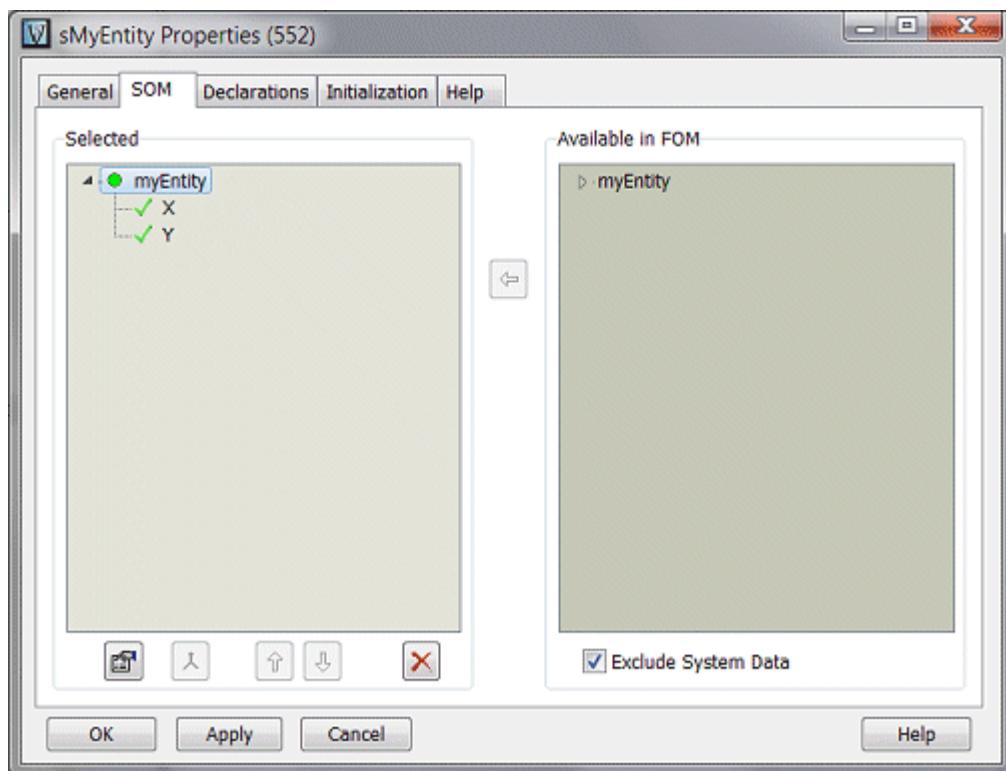
## Subscribe

# Subscribe

### • Subscriber Object

Let's add the `myEntity` subscriber object.

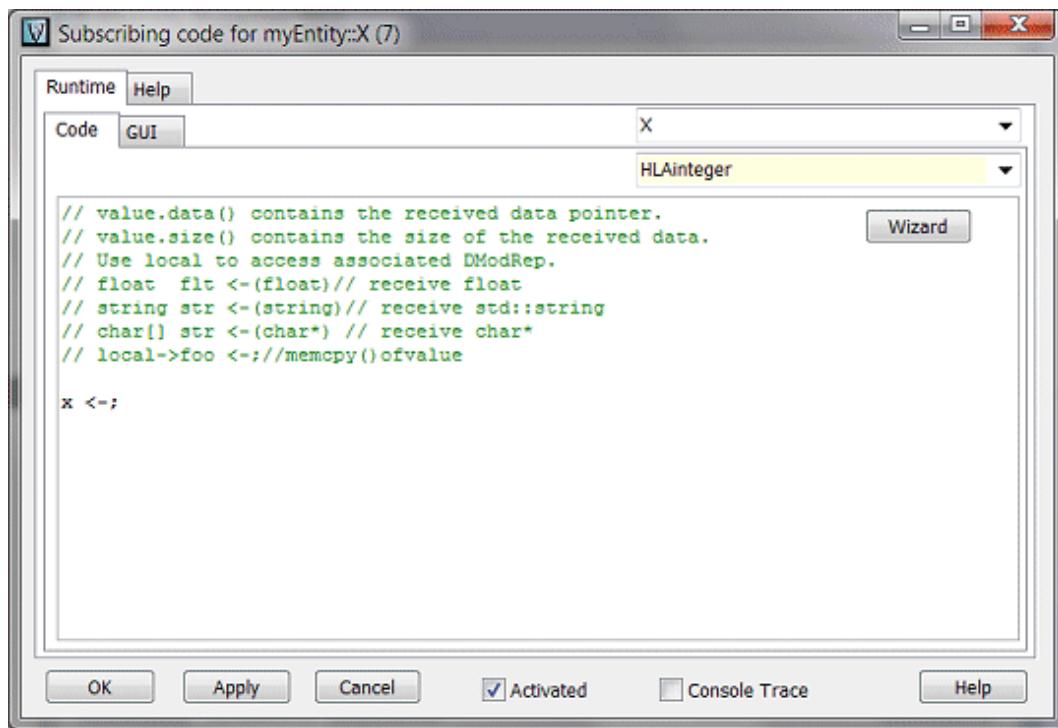
It will be called after each `myEntity` object that publish on the Federation.



In the panel Declaration, let's define the two local variables that will hold the values coming from the RTI.

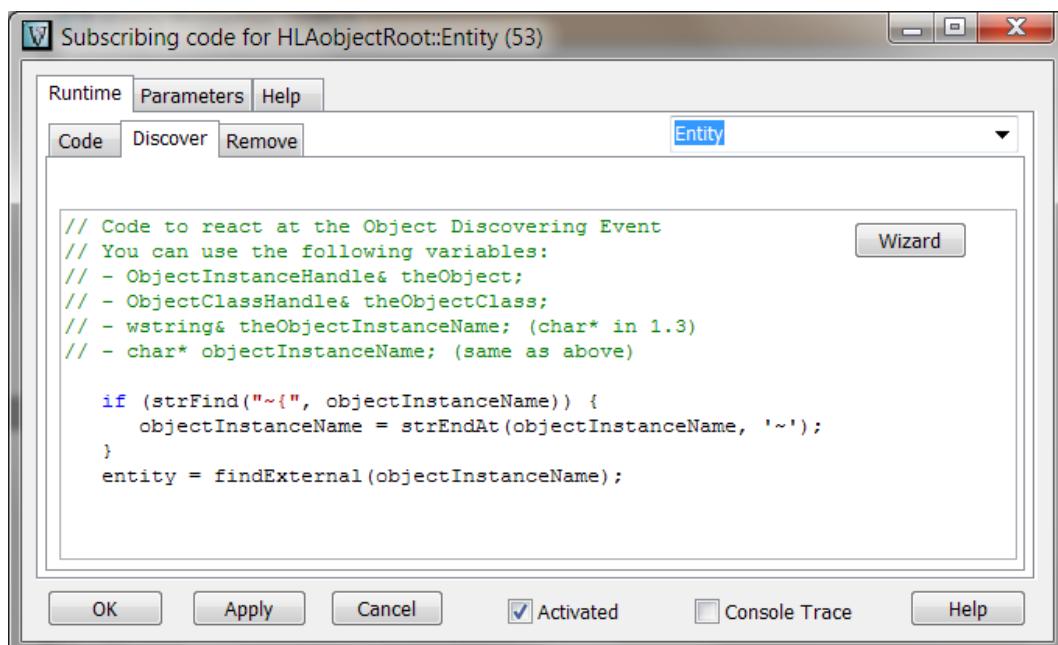
```
int x, y;
```

Then, for each Attribute, let's make the copy:



X and Y must be copied to local variables `x` and `y`, then `myEntity` Object code will be called.

There, we will update the entity position.



First time the Object is found (new handle), the Discover part is called.

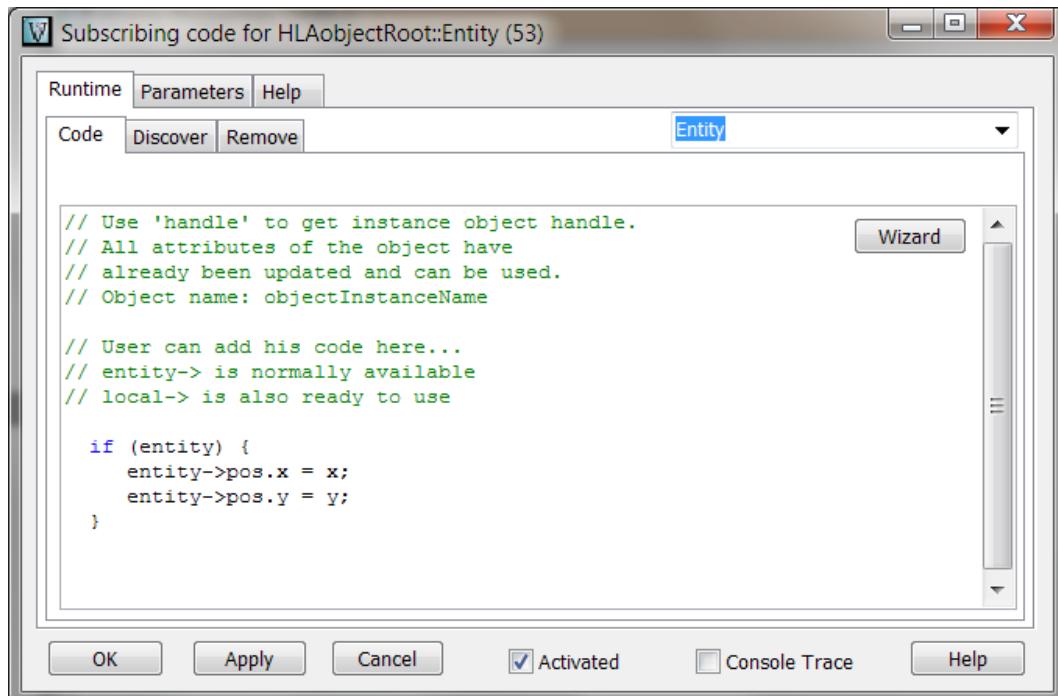
If the option is checked, an entity will be created first and paired with the handle.

## Subscribe

If you do not want to create a default entity, the correct typed one must be created in the Discover panel.

```
entity = new Entity("myEntityType", objectInstanceName);
add(entity); // in the subscriber object list
store(entity); // and associate it with handle
```

When the Entity is created or retrieved at next updates, setting is done:



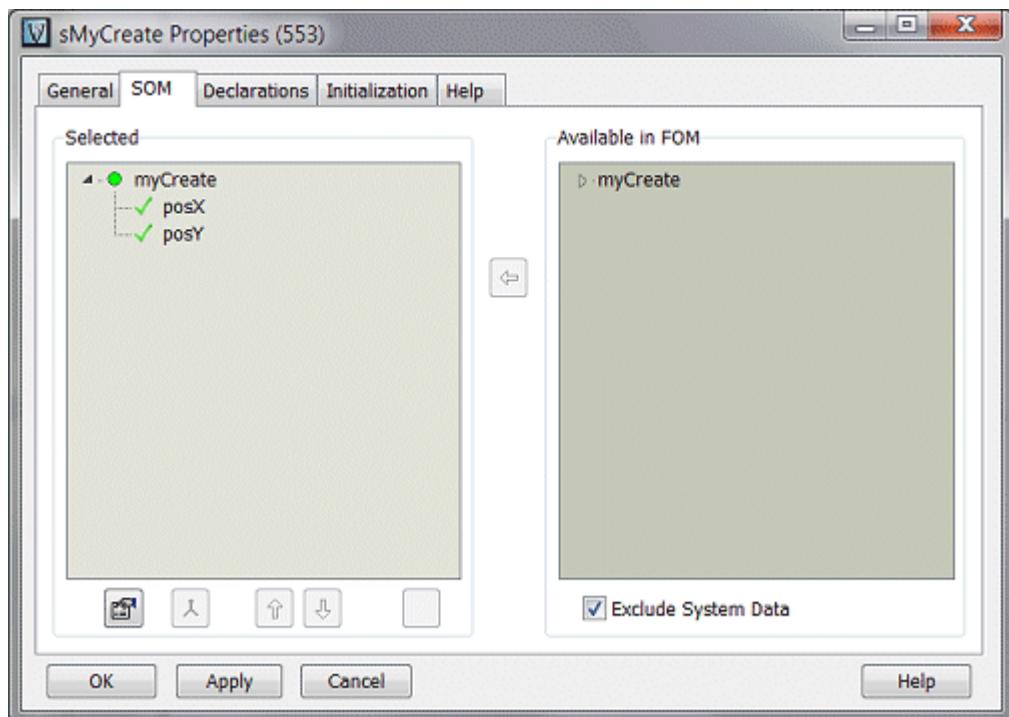
```
entity->pos.x = x;
entity->pos.y = y;
```

## • Subscriber Interaction

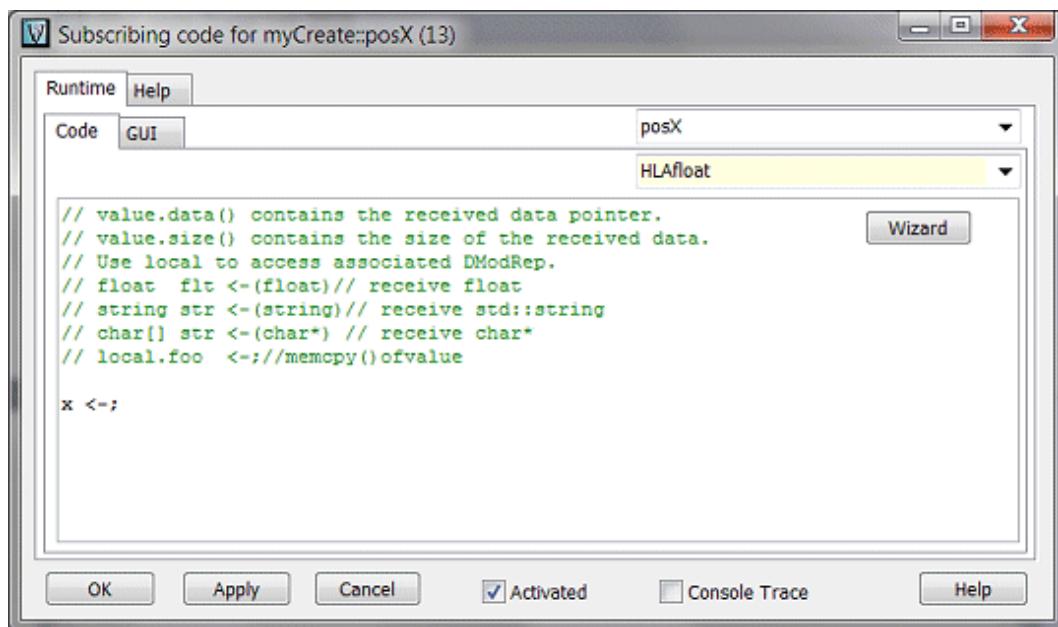
The [myCreate](#) interaction will receive the mouse position.

We will declare the two variables that will keep the data coming from the RTI, using the *Declaration* panel:

```
float x, y;
```

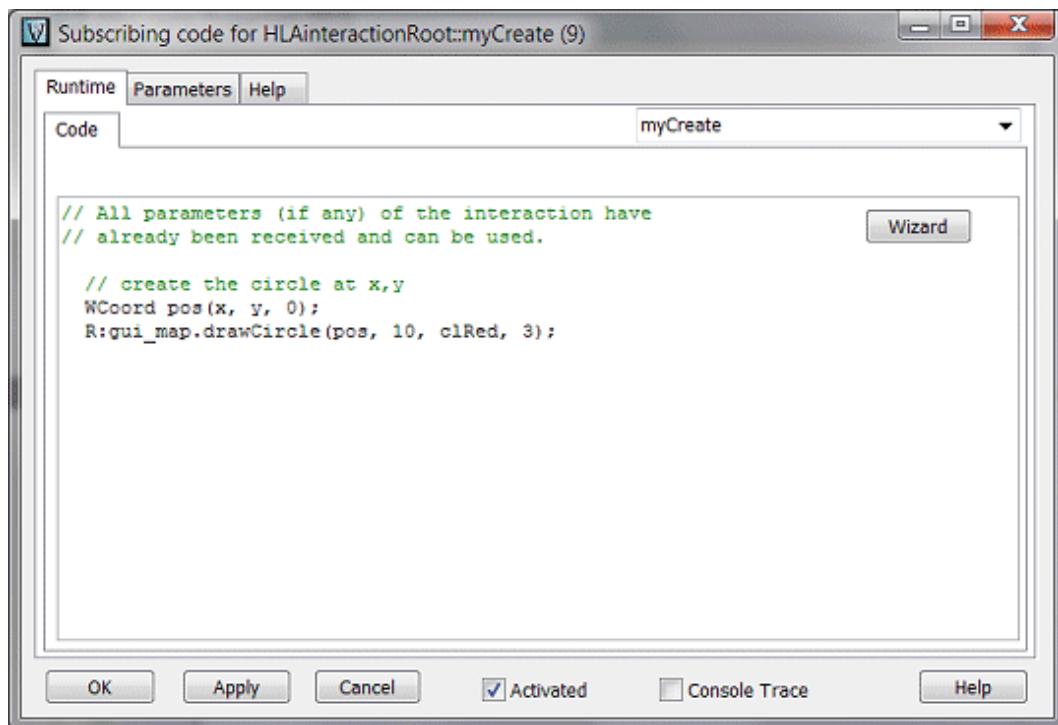


We will now copy the `posX` parameter into the `x` local variable (and do the same with `posY`).



Now, in the `myCreate` Interaction code, let's use the `(x,y)` map coordinates to draw a red circle for 3 seconds.

## Subscribe



```
// create the circle at x,y
WCoord pos(x, y, 0);
R:gui_map.drawCircle(pos, 10, clRed, 3);
```

Time now to [run the simulation...](#)

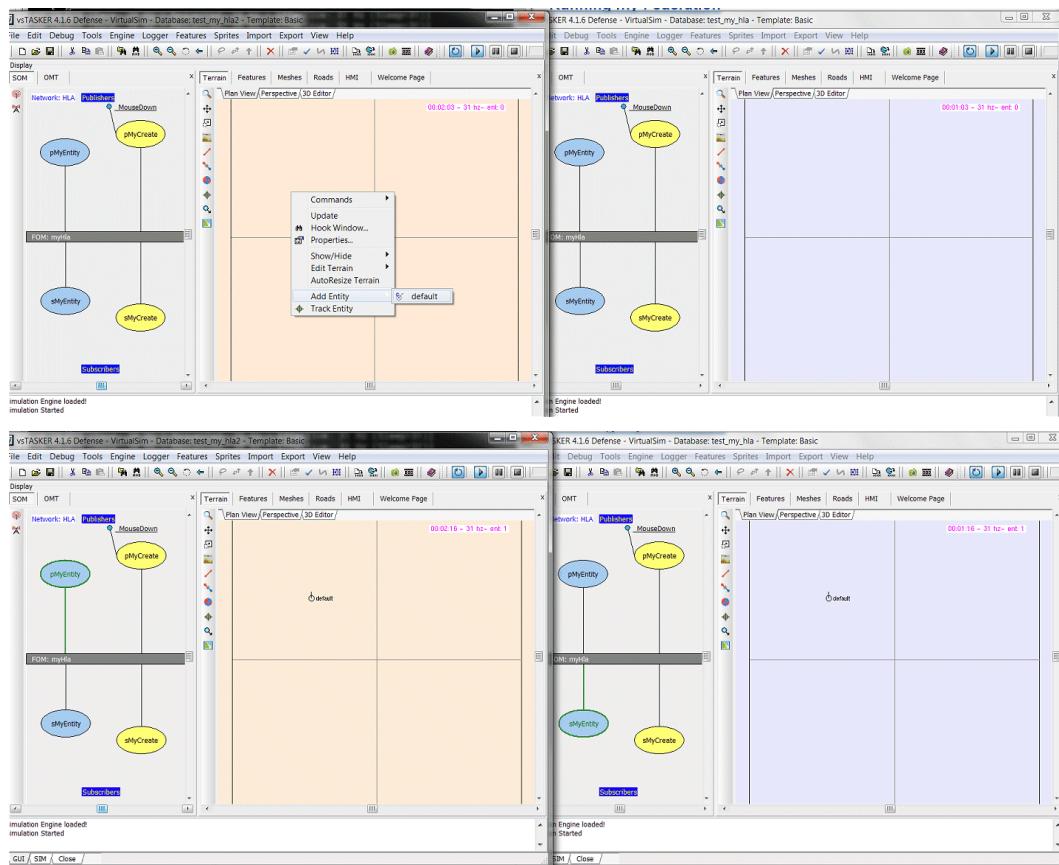
# Running my Federation

For building the second vsTASKER, refer to [this description](#).

If you have configured the RTI to be used on two computers, you can then start the blue database on one and the red database on the other.  
Then, start both simulations.

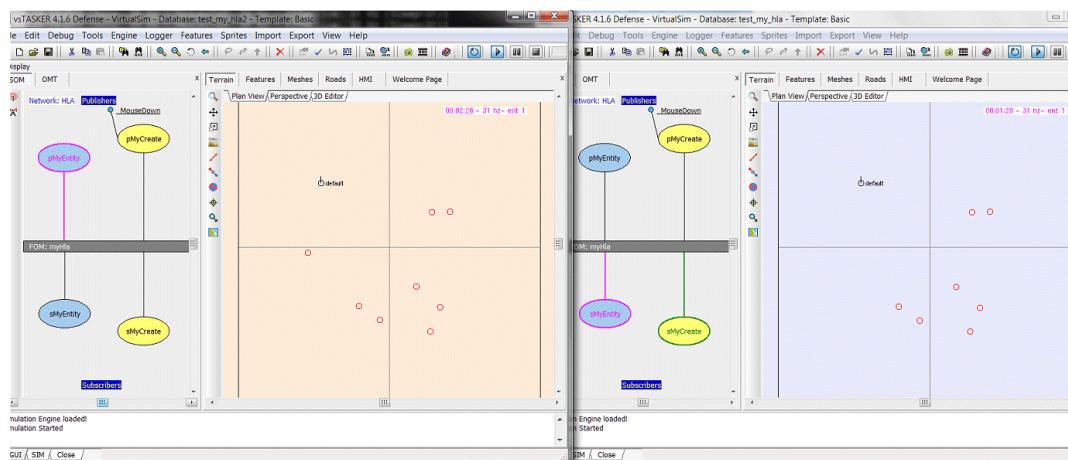
Creating (or removing) one entity in either blue or red vsTASKER will distribute the Entity (as external) on the other.

Finally, relocate any local Entity to the map and see how it moves on the other.



Once both simulations are running, check that clicking on the map of one vsTASKER will draw a red circle on both.

## Running my Federation



# **Complex Federations**

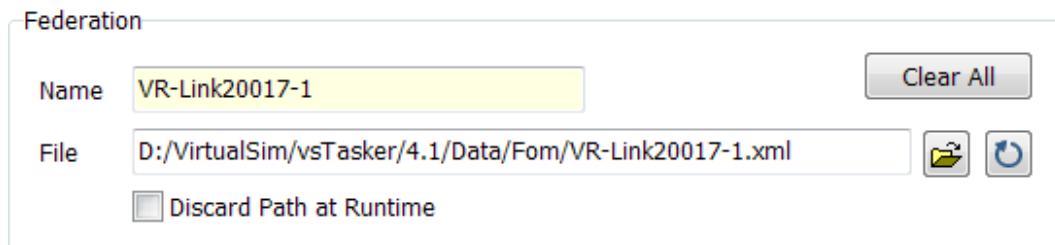
Here, we will see how to handle big and complex Federations.

## RPR-FOM

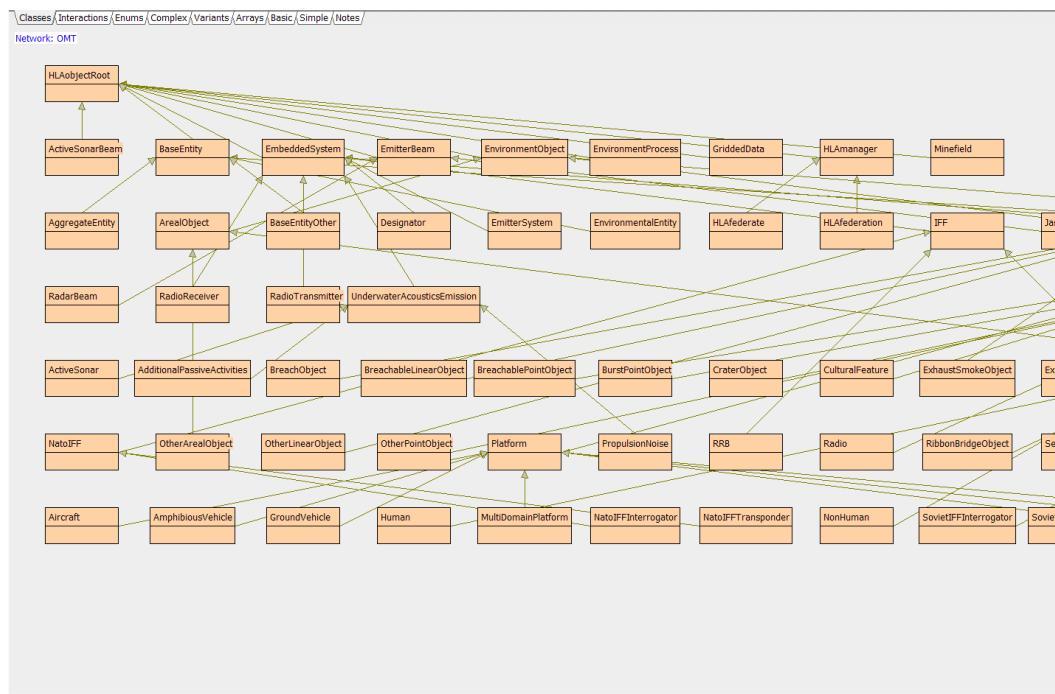
# RPR-FOM

In this demo, we will use the RPR FOM to connect to a sample federate provided by Mak Technology (F18HLA1516.ex) with VR-Link.  
We will subscribe to an [Aircraft](#) entity (F16) and its [Emitter Beam](#).  
We will also publish our own [Aircraft](#).  
[Fire/Detonation](#) will also be managed.

First, let's open the FOM



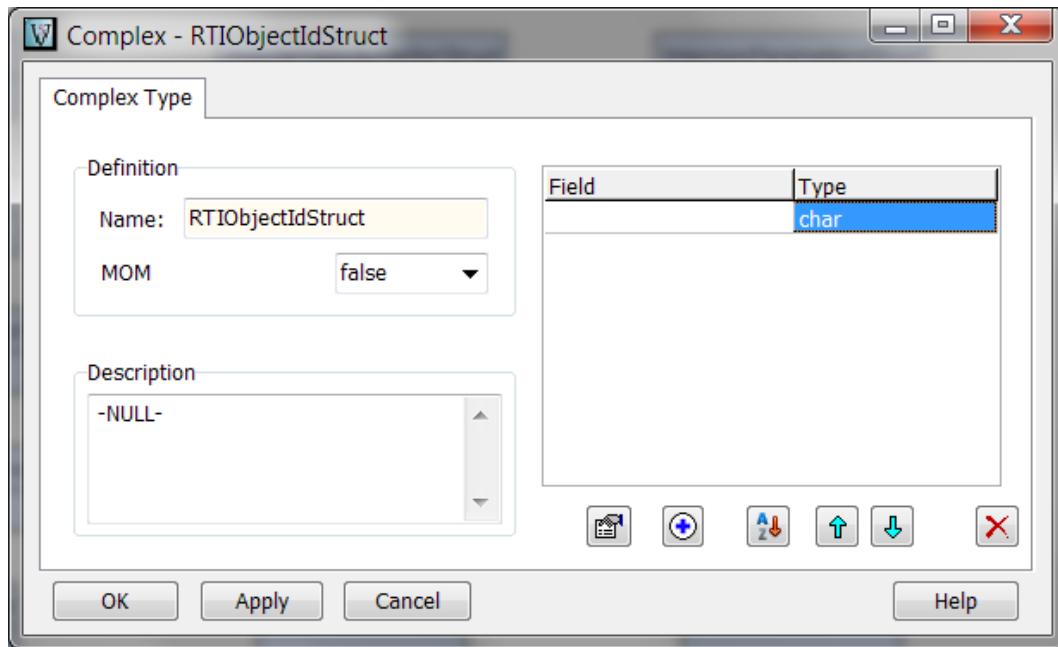
In the OMT Part, we can see that the FOM has been converted in local editable objects:



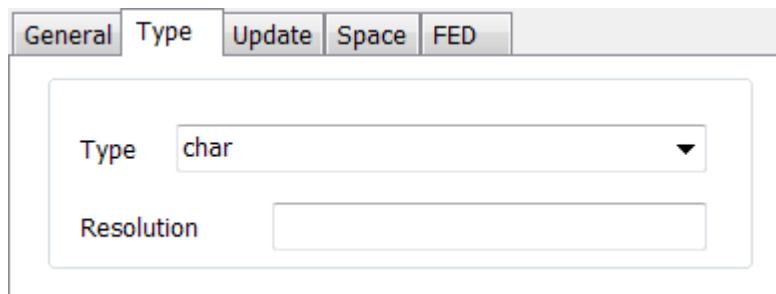
When the hierarchy is complex, it could be nice to sort things. As there is no tool to automate that, use the [Diagram](#) mode to maximum size then move any object with **shift** key depressed to also move all its children.

You can also right click the mouse and use the context menu [Gather Below](#) to put all the children recursively below each selected parent in order to visually and manually rearrange the whole OMT hierarchy.

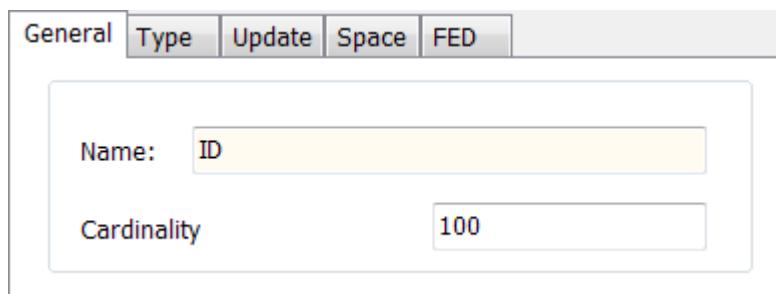
We can change any of them before converting to local C++ data structures.  
In the Complex panel, let's Edit the [RTIObjecIdStruct](#):



Let's Edit the **ID** Field by changing the **Type** to [char](#):



and the **Cardinality** to [100](#) (or more)



- **Making the SOM**

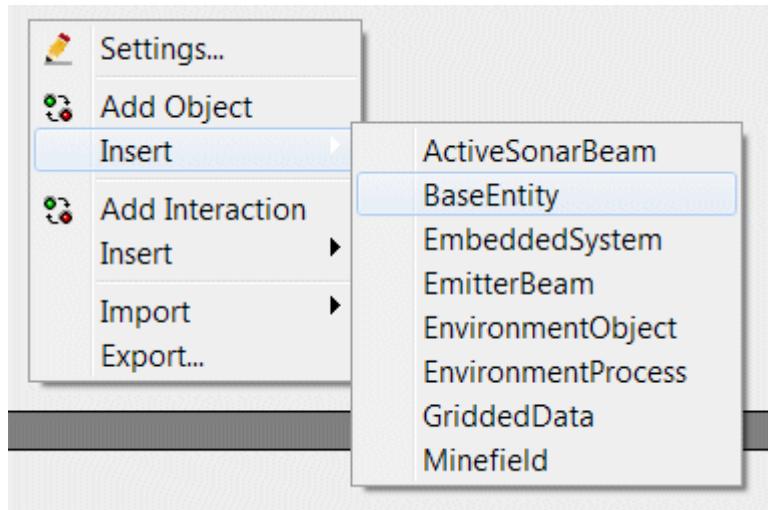
## RPR-FOM

In the SOM panel, let's just add the Objects and Interactions we want to Subscribe and Publish.

We will Insert the following Objects:

**Publisher:** Aircraft

**Subscriber:** BaseEntity, EmitterSystem and EmitterBeam.



Aircraft Object belongs to BaseEntity. So, we need to insert it from the Publisher area.

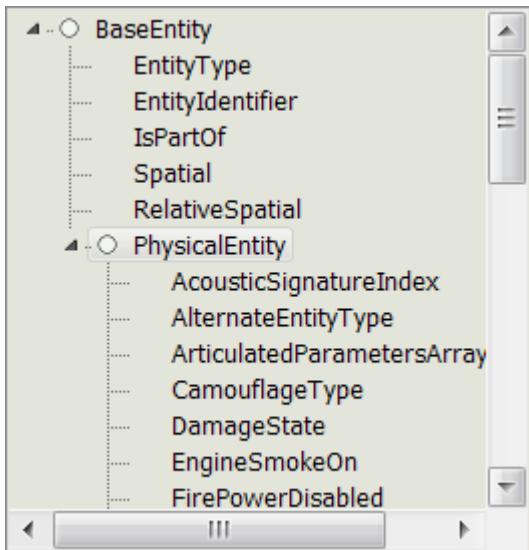
Let's now open BaseEntity FeedItem, let's rename it to pAircraft (p for Publish) and let's select the Aircraft Object by merging it with its parent nodes.

A screenshot of the SOM panel showing the 'Selected' and 'Available in FOM' panes. The 'Selected' pane on the left shows a tree view of the BaseEntity node and its children: EntityType, EntityIdentifier, IsPartOf, Spatial, RelativeSpatial, BaseEntityOther, AggregateEntity, EnvironmentalEntity, and PhysicalEntity. The 'Available in FOM' pane on the right lists the same objects as the secondary menu in the previous image: ActiveSonarBeam, BaseEntity, EmbeddedSystem, EmitterBeam, EnvironmentObject, EnvironmentProcess, GriddedData, and Minefield. At the bottom of the 'Available in FOM' pane, there is a checked checkbox labeled 'Exclude System Data'.

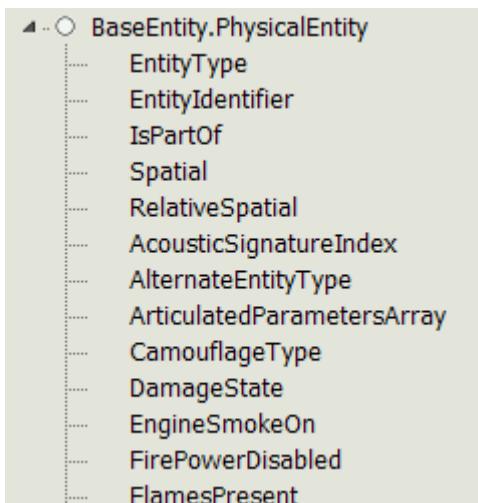
To merge [PhysicalEntity](#) with  [BaseEntity](#), we must remove all Objects in between, here,  [BaseEntityOther](#),  [AggregateEntity](#) and  [EnvironmentalEntity](#).

If we need to subscribe to them, we will create another FedItem for each of them.

Select each of them from the list then press  (if many, use multiple selections, using the shift or ctrl key) until you get:



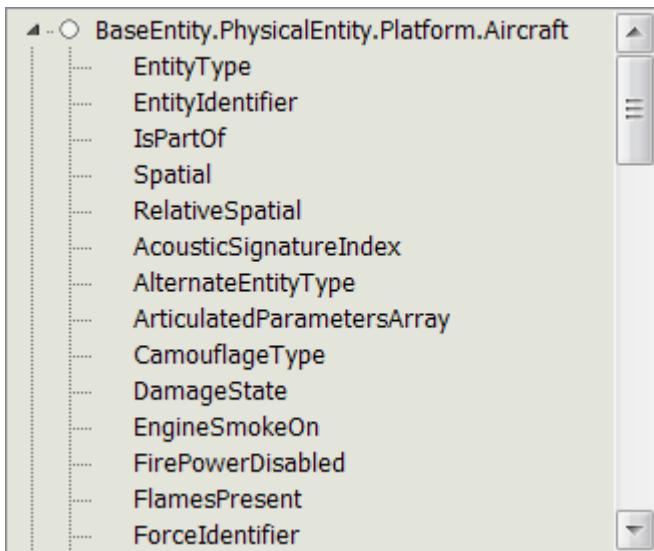
Now, select [PhysicalEntity](#) Object and merge it with parent ( [BaseEntity](#)) using   
You will get the merged new Object:



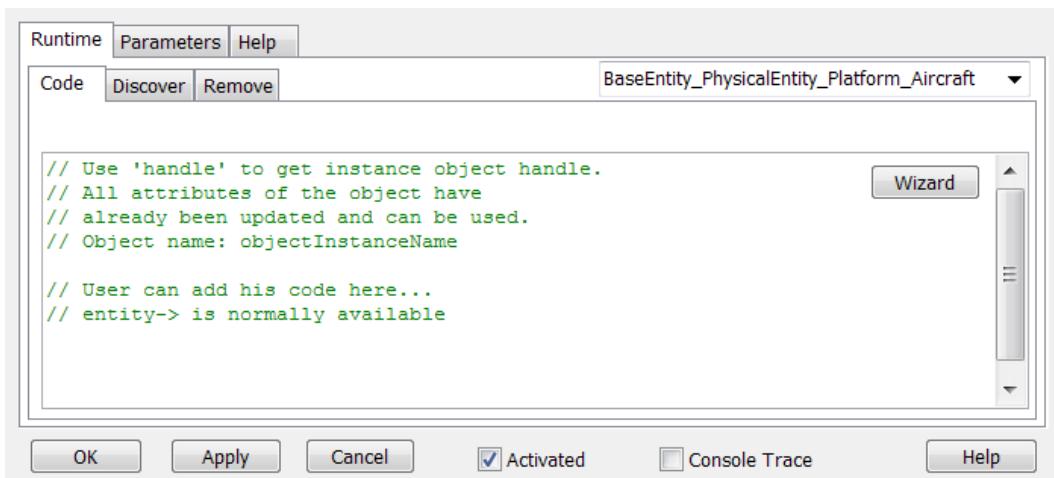
At the bottom of the list, select Object [Platform](#) then use 

Finally, let's do the same for [Aircraft](#) Object at the bottom of the list.  
You should get the following:

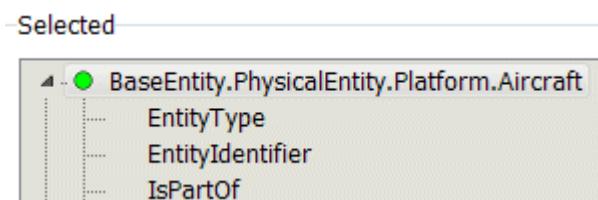
## RPR-FOM



We will now select the  `BaseEntity.PhysicalEntity.Platform.Aircraft`  Object by opening it and just check the [Activate](#) checkbox:

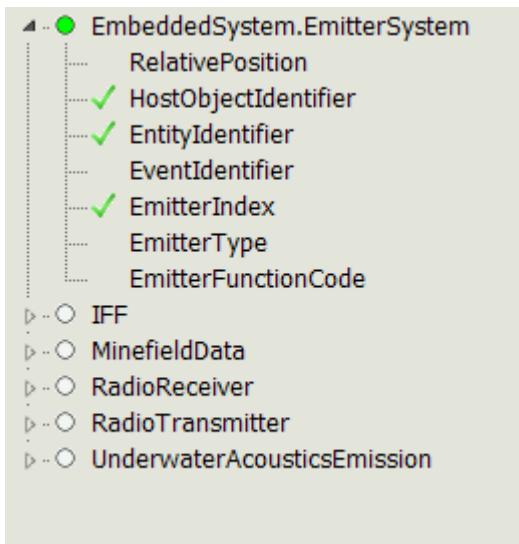


Now, the green lamp shows that the Item is selected for publishing:



In the Subscriber side, let's just add  [BaseEntity](#)  Object, then  [EmitterBeam](#)  and the  [EmbeddedSystem](#) .

For the  [EmbeddedSystem](#) , the  [EmitterSystem](#)  Object must be merged because we need some of its attributes:



You do it the same way as with [Aircraft](#).

Other Objects can remain in the list as long as they are not Activated. They will be ignored by the code generator.

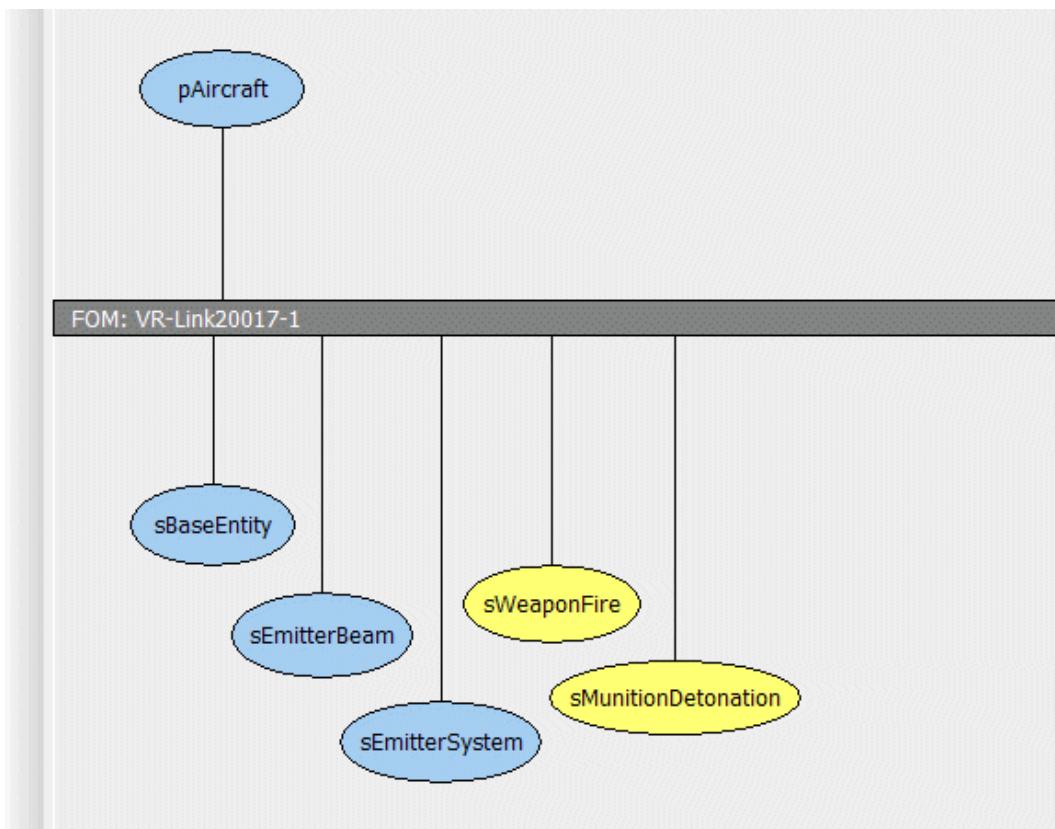
Do not forget to activate the base Object of each FedItem.

Finally, add the two Interactions: [WeaponFire](#) and [MunitionDetonation](#).

Activate them also.

You should then have the following:

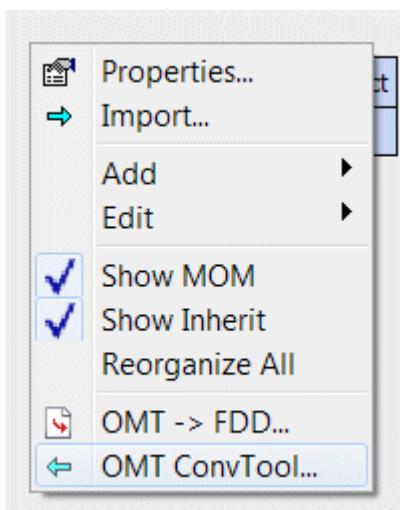
## RPR-FOM



## • Making the DataModels

In order to simplify the data manipulation (from and to the RTI), we will use the [OMT -> DataModel](#) converter.

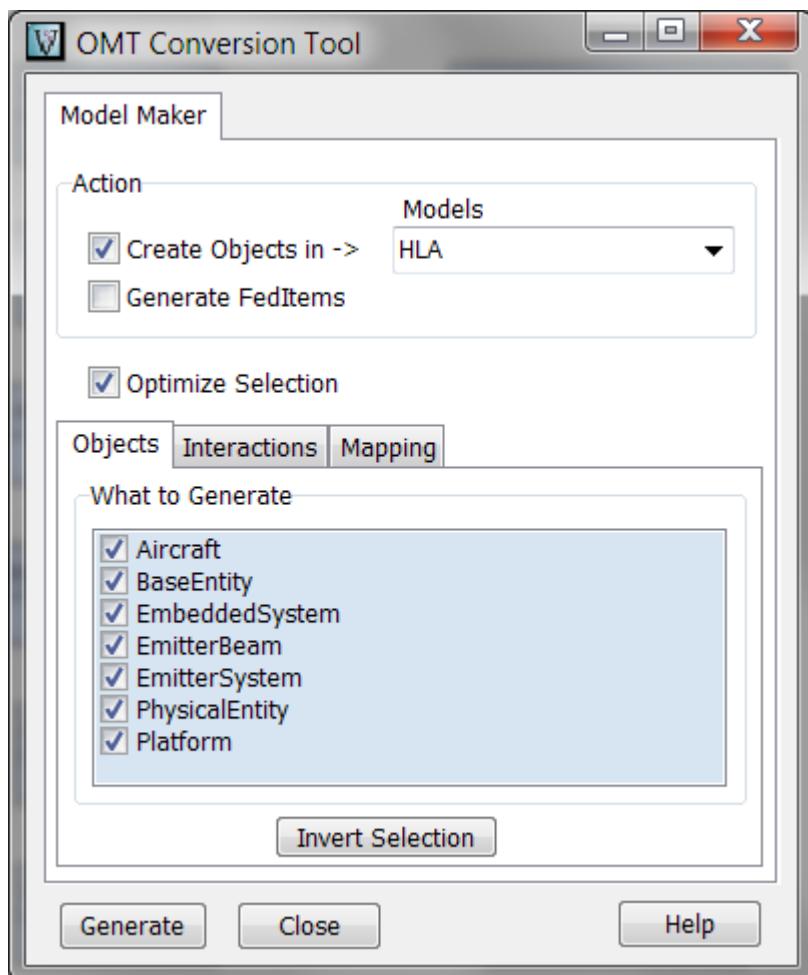
Let's go to the OMT panel, then right click to select the [OMT ConvTool...](#):



Select "Optimize Selection" to filter only the Objects/Interactions (and inherited ones) as well as used other data structures to be converted as DataModels.

There is no need to translate the full OMT database as only some Objects will be needed and used.

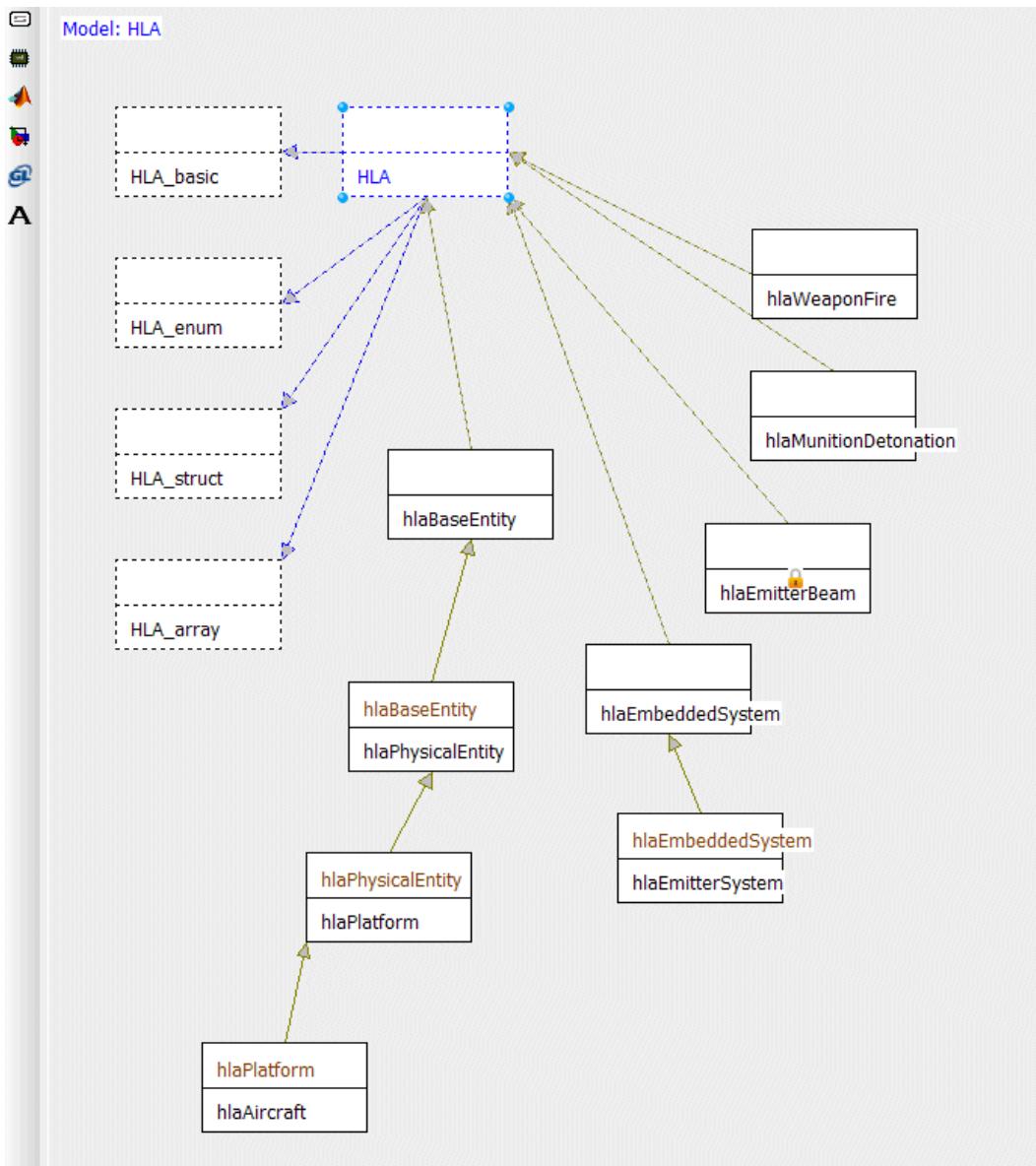
If later, some must be added, they just can be manually added.



Now press **Generate**.

Back in the Hla Model diagram view, you should see something like that:

## RPR-FOM



## • Subscribing to BaseEntity

Back again to the SOM view, time to fill the **s BaseEntity** FeedItem.  
First, we need to associate a DataModel. We will use the **hlaBaseEntity** one:

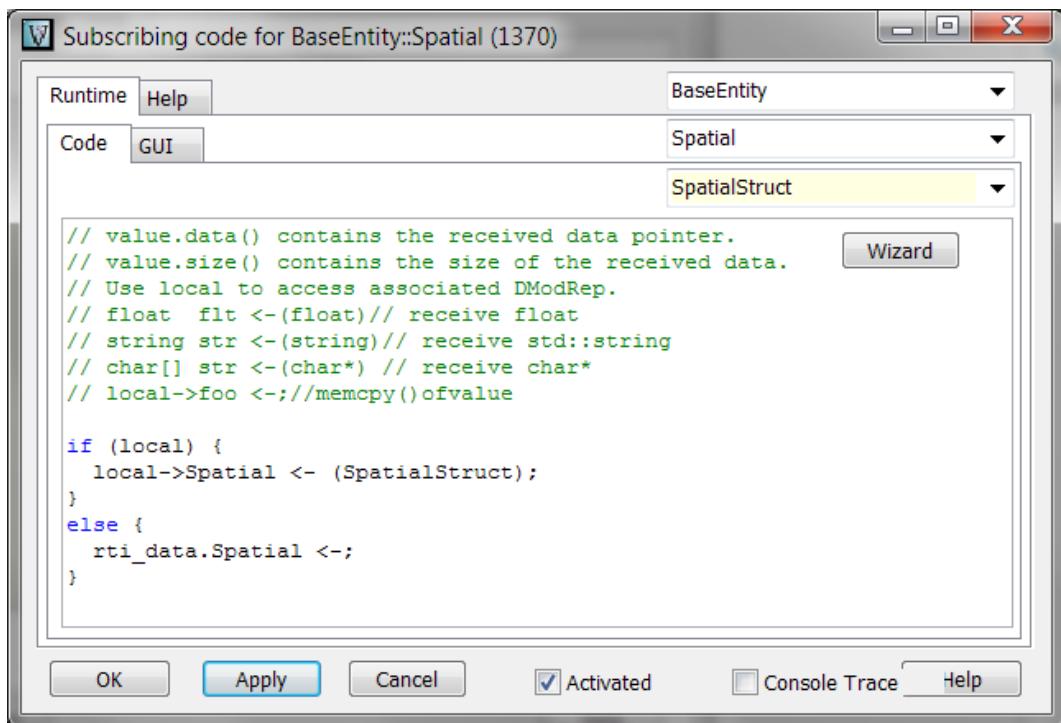
Modeling

DataModel:	<b>hlaBaseEntity</b>
------------	----------------------

Because the Object is associated with an Entity, we need to check the  Allow RT Entities checkbox. This will insure proper code generation related to Entity creation itself.

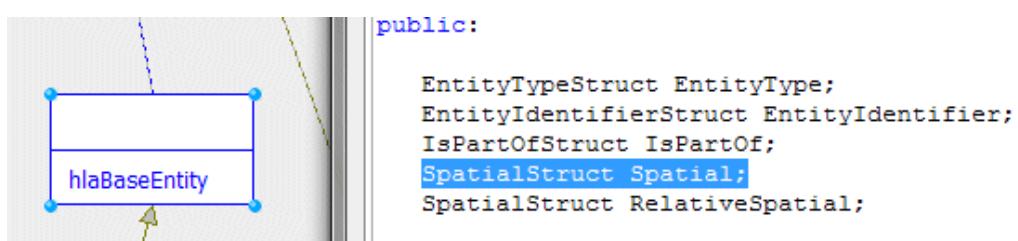
If not checked, user will need to handle manually Entity new, delete and handle pairing.

Now, we will edit the [Spatial](#) Attribute and use the *Wizard* button:



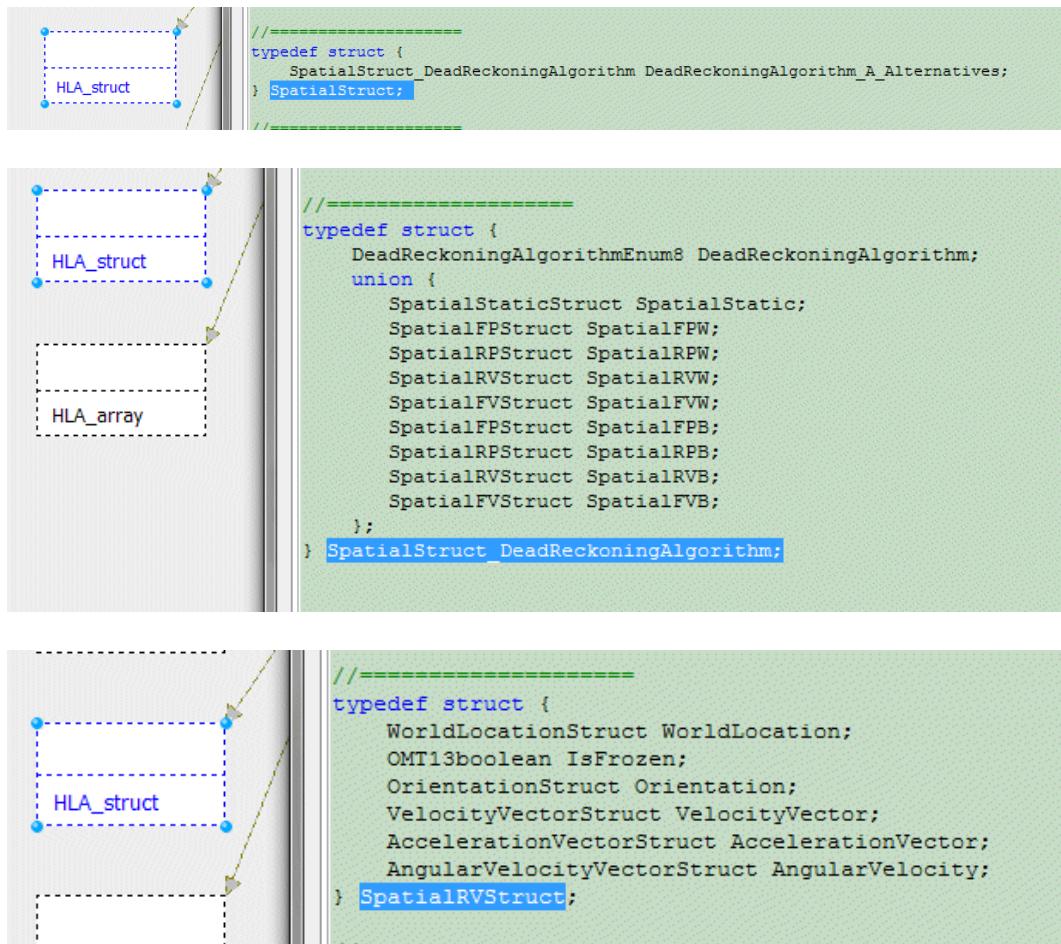
The code means that `local` variable pointer will change the DataModel associated with the Entity (if any), otherwise, the `rti_data` structure will be used instead. Both `rti_data` and `local` are automatically generated and of type of the DataModel used (here: [hlaBaseEntity](#)).

To know the structure definition of [Spatial](#), we can either use the OMT definition or the DataModel definitions:



Then:

## RPR-FOM



We need to convert to Big Endian all coming data from the RTI, fields by fields.  
Let's do the following into the [Spatial](#) Attribute:

```
if (local) {
    local->Spatial <- (SpatialStruct);

    SpatialRVStruct& data = &local-
>Spatial.DeadReckoningAlgorithm_A_Alternatives.SpatialRVW;

    ENDIAN_SWAP_DOUBLE64(data.WorldLocation.X);
    ENDIAN_SWAP_DOUBLE64(data.WorldLocation.Y);
    ENDIAN_SWAP_DOUBLE64(data.WorldLocation.Z);
    ENDIAN_SWAP_FLOAT32(data.Orientation.Psi);
    ENDIAN_SWAP_FLOAT32(data.Orientation.Theta);
    ENDIAN_SWAP_FLOAT32(data.Orientation.Phi);
    ENDIAN_SWAP_FLOAT32(data.VelocityVector.XVelocity);
    ENDIAN_SWAP_FLOAT32(data.VelocityVector.YVelocity);
    ENDIAN_SWAP_FLOAT32(data.VelocityVector.ZVelocity);
    ENDIAN_SWAP_FLOAT32(data.AngularVelocity.XAngularVelocity);
```

```

    ENDIAN_SWAP_FLOAT32(data.AngularVelocity.YAngularVelocity);
    ENDIAN_SWAP_FLOAT32(data.AngularVelocity.ZAngularVelocity);
}

```

Let's do similar for [EntityIdentifier](#) Attribute:

```

if (local) {
    local->EntityIdentifier <- (EntityIdentifierStruct);

    ENDIAN_SWAP_SHORT16(local->EntityIdentifier.FederateIdentifier.SiteID);
    ENDIAN_SWAP_SHORT16(local-
>EntityIdentifier.FederateIdentifier.ApplicationID);
    ENDIAN_SWAP_SHORT16(local->EntityIdentifier.EntityNumber);
}

```

... and [EntityType](#) Attribute:

```

if (local) {
    local->EntityType <- (EntityTypeStruct);

    ENDIAN_SWAP_SHORT16(local->EntityType.CountryCode);
}

```

Now, in the  [BaseEntity](#) code, you can write the following, to process the incoming data and update the corresponding vsTASKER data structures and Models:

```

SpatialRVStruct& data = &local-
>Spatial.DeadReckoningAlgorithm_A_Alternatives.SpatialRVW;

WCoord pos(WC_ECEF, data.WorldLocation.X, data.WorldLocation.Y,
data.WorldLocation.Z);
pos.convertECEFtoXYZ();
entity->pos = pos;

pos.convertXYZtoLLA();
pos.setOrigin(pos.lat, pos.lon);
Vec3d& hpr = pos.eulerToHpr(data.Orientation.Psi, data.Orientation.Theta,
data.Orientation.Phi);
entity->getDyn()->setHeading(-hpr[0]);

Vec3d& spd = pos.remoteToLocal(data.VelocityVector.XVelocity,
data.VelocityVector.YVelocity, data.VelocityVector.ZVelocity);
float s = sqrt(SQR(spd[0])+SQR(spd[1])+SQR(spd[2]));
entity->getDyn()->setSpeed(s);

```

## RPR-FOM

```
data.AngularVelocity.XAngularVelocity = 0;
data.AngularVelocity.YAngularVelocity = 0;
data.AngularVelocity.ZAngularVelocity = 0;

PtfDisType& type = *(PtfDisType*) entity->findComponent("PtfDisType"); // this
component belongs to MillLib
if (type) {
    type.kind      = local->EntityType.EntityKind;
    type.domain    = local->EntityType.Domain;
    type.country   = local->EntityType.CountryCode;
    type.category  = local->EntityType.Category;
    type.subcategory = local->EntityType.Subcategory;
    type.specific   = local->EntityType.Specific;
    type.extra      = local->EntityType.Extra;
}
else
printf("Warning: %s does not have %s DataModel!\n", entity->getName(),
"PtfDisType");

PtfStatus* status = (PtfStatus*) entity->findDataModel("PtfStatus");
if (status) {
    type->update(status);
    status->updateSymbol(SM_2525B);
}
else printf("Warning: %s does not have %s DataModel!\n", entity->getName(),
"PtfStatus");
```

## • Publishing Aircraft

Let's open the [pAircraft](#) Object to edit the [Spatial](#) Attribute. We first use the *Wizard* button to get the code and then, we add the Endian swap macro after the receiving from the RTI:

```

// if attribute name is "foo", assign value like this:
// 1516: "foo" <- (float) &foo;
//           "foo" <- (string) foo;
// 1.3:   "foo" <- (char*) sbuf; // char sbuf[250];
// All:   "foo" <- (&foo, sizeof(float)); // float foo;

if (local) {
    SpatialRVStruct& data = local->Spatial.DeadReckoningAlgorithm_A_Alternatives.SpatialRVW;

    ENDIAN_SWAP_DOUBLE64(data.WorldLocation.X);
    ENDIAN_SWAP_DOUBLE64(data.WorldLocation.Y);
    ENDIAN_SWAP_DOUBLE64(data.WorldLocation.Z);
    ENDIAN_SWAP_FLOAT32(data.Orientation.Psi);
    ENDIAN_SWAP_FLOAT32(data.Orientation.Theta);
    ENDIAN_SWAP_FLOAT32(data.Orientation.Phi);
    ENDIAN_SWAP_FLOAT32(data.VelocityVector.XVelocity);
    ENDIAN_SWAP_FLOAT32(data.VelocityVector.YVelocity);
    ENDIAN_SWAP_FLOAT32(data.VelocityVector.ZVelocity);
    ENDIAN_SWAP_FLOAT32(data.AngularVelocity.XAngularVelocity);
    ENDIAN_SWAP_FLOAT32(data.AngularVelocity.YAngularVelocity);
    ENDIAN_SWAP_FLOAT32(data.AngularVelocity.ZAngularVelocity);

    "Spatial" <- (SpatialStruct) &local->Spatial;
}

```

We do the same for the [EntityIdentifier](#) Attribute:

```

if (local) {
    ENDIAN_SWAP_SHORT16(local->EntityIdentifier.FederateIdentifier.SiteID);
    ENDIAN_SWAP_SHORT16(local-
>EntityIdentifier.FederateIdentifier.ApplicationID);
    ENDIAN_SWAP_SHORT16(local->EntityIdentifier.EntityNumber);

    "EntityIdentifier" <- (EntityIdentifierStruct) &local->EntityIdentifier;
}

```

... and the [EntityType](#) one:

```

if (local) {
    ENDIAN_SWAP_SHORT16(local->EntityType.CountryCode);

    "EntityType" <- (EntityTypeStruct) &local->EntityType;
}

```

Then, at the Object code level, we need to set the data first.

A Publisher FedItem is called for all Entities (or Handles) it holds.

When the FedItem is Entity Based ( [Allow RT Entities](#) ), it processes [used\\_by](#) entities one by one and retrieve the associated Handle. When the FedItem is Handle based ( [Allow RT Entities](#) ), it processes Handles one by one and retrieve the associated Entity (if any).

## RPR-FOM

```
if (ent_idx >= getEntities().count()) { ent_idx = 0; return LEAVE; }
else {
    entity = getEntity(ent_idx++);
    local = entity? (hlaAircraft*)entity->findDataModel("hlaAircraft"): NULL;
    // User can add his code here...

    if (local) {
        SpatialRVStruct& data = local-
>Spatial.DeadReckoningAlgorithm_A_Alternatives.SpatialRVW;

        double h = entity->getDyn()->getHeading();
        double p = entity->getDyn()->getElev();
        double r = entity->getDyn()->getRoll();

        // position
        WCoord pos(WC_XYZ, entity->pos.x, entity->pos.y, entity->pos.z);
        pos.convertXYZtoECEF();
        data.WorldLocation.X = pos.x;
        data.WorldLocation.Y = pos.y;
        data.WorldLocation.Z = pos.z;

        // orientation
        pos.convertECEFtoLLA();
        pos.setOrigin(pos.lat, pos.lon);
        Vec3d& vec = pos.hprToEuler(-h,p,r);
        data.Orientation.Psi    = vec[0];
        data.Orientation.Theta = vec[1];
        data.Orientation.Phi   = vec[2];

        // speed
        double s = entity->getDyn()->getSpeed();
        Vec3d& spd = pos.localToRemote(s*sin(h)*cos(p), s*cos(h)*cos(p),
s*sin(p));
        data.VelocityVector.XVelocity = spd[0];
        data.VelocityVector.YVelocity = spd[1];
        data.VelocityVector.ZVelocity = spd[2];

        // identifier
        local->EntityIdentifier.FederateIdentifier.SiteID = 1;
        local->EntityIdentifier.FederateIdentifier.ApplicationID = 1;
        local->EntityIdentifier.EntityNumber = entity->getId();

        // DIS type
        PtfDisType& type = *(PtfDisType*) entity->findComponent("PtfDisType");
```

```

    if (type) {
        local->EntityType.EntityKind = type.kind;
        local->EntityType.Domain = type.domain;
        local->EntityType.CountryCode = type.country;
        local->EntityType.Category = type.category;
        local->EntityType.Subcategory = type.subcategory;
        local->EntityType.Specific = type.specific;
        local->EntityType.Extra = type.extra;
    }
    else printf("Warning: %s does not have %s DataModel!\n", entity-
>getName(), "PtfDisType");
}
return CONTINUE;
}
return PROCEED;

```

## • Emitter System

When 'e' key is depressed in the F18.exe window, an [EmitterSystem](#) Object is created and an [EmitterBeam](#) Object is also attached to it.

Both Objects are related to an Entity but one [EmitterSystem](#) can hold several [EmitterBeam](#) while one Entity normally got one [EmitterSystem](#).

The [EmitterSystem](#) FedItem is not Entity based ( Allow RT Entities ) so, Handles must be paired manually with any Entity and extra `void*` data that might be needed. This is handy because next time the Handle will reflect updates, Entity and user data pointer will be returned back.

In Subscriber FedItems, Attributes/Parameters are received first, then the Object/Interaction code is called.

In our case, the Discover panel is of no use because we do not know yet what to do with the Handle (we do not create an Entity).

So, it will be in the [EmitterSystem](#) Object code that we will pair the Handle with the referred Entity:

```

// No entity paired yet with the Handle.
if (!entity) {
    // We must look for an external entity whose name is given by
the ID
    entity = S:findEntity(rti_data.HostObjectIdentifier.ID);
    // if one is found...
    if (entity) {

```

## RPR-FOM

```
        add(entity); // we add it to the used_by list of the
FedItem, for tracability mainly
        store(entity); // we associate it with the handle. Next
time, we will not come here
    }
}
```

When Entity is found, we can then retrieve the Entity DataModel and set the values coming from the RTI.

```
if (entity) local = (hlaEmitterSystem*) entity-
>findComponent("hlaEmitterSystem");
if (local) {
    local->EntityIdentifier = rti_data.EntityIdentifier;
    local->HostObjectIdentifier = rti_data.HostObjectIdentifier;
    local->EmitterIndex = rti_data.EmitterIndex;
}
```

## • Emitter Beam

For this Object, we need to get the following Attributes:

**EmitterSystemIdentifier**: name of the RTI Object. With it, we will retrieve the associated Handle and then, the Entity:

```
if (local) {
    local->EmitterSystemIdentifier <- (RTIOBJECTIDSTRUCT);
}
else {
    rti_data.EmitterSystemIdentifier <-;
}
```

**EffectiveRadiatedPower**: just for the length of the beam:

```
if (local) {
    local->EffectiveRadiatedPower <- (HLAfloat32BEEdBmperfectalways);
    ENDIAN_SWAP_FLOAT32(local->EffectiveRadiatedPower);
}
else {
    rti_data.EffectiveRadiatedPower <-;
    ENDIAN_SWAP_FLOAT32(rti_data.EffectiveRadiatedPower);
}
```

**BeamAzimuthSweep:** for the beam wideness value :

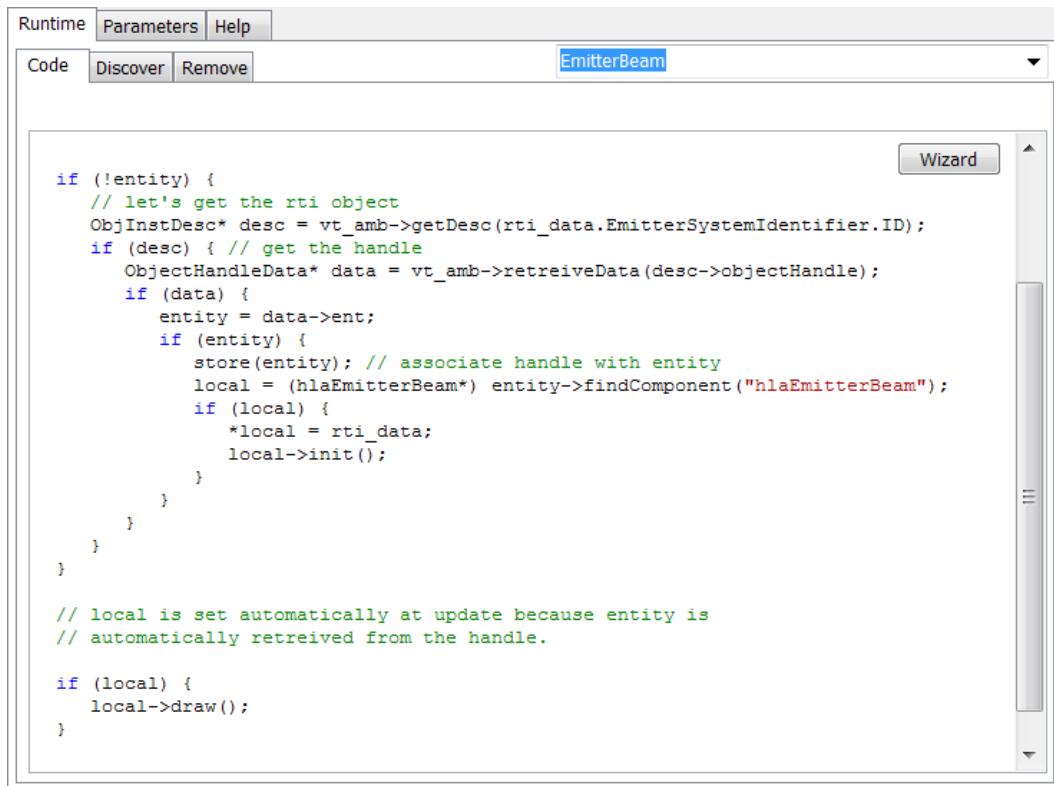
```
if (local) {
    local->BeamAzimuthSweep <- (HLAfloat32BEradiansperfectalways);
    ENDIAN_SWAP_FLOAT32(local->BeamAzimuthSweep);
}
else {
    rti_data.BeamAzimuthSweep <-;
    ENDIAN_SWAP_FLOAT32(rti_data.BeamAzimuthSweep);
}
```

**BeamAzimuthCenter:** for the beam current azimuth :

```
if (local) {
    local->BeamAzimuthCenter <- (HLAfloat32BEradiansperfectalways);
    ENDIAN_SWAP_FLOAT32(local->BeamAzimuthCenter);
}
else {
    rti_data.BeamAzimuthCenter <-;
    ENDIAN_SWAP_FLOAT32(rti_data.BeamAzimuthCenter);
}
```

Once that done, we just need now to set the DataModel according to the coming values:

## RPR-FOM



The screenshot shows the vsTASKER 7 RPR-FOM interface. At the top, there are tabs for Runtime, Parameters, Help, Code, Discover, and Remove. The Discover tab is selected. In the center, the object name 'EmitterBeam' is displayed. On the right, there is a 'Wizard' button. The main area contains the following C++ code:

```
if (!entity) {
    // let's get the rti object
    ObjInstDesc* desc = vt_amb->getDesc(rti_data.EmitterSystemIdentifier.ID);
    if (desc) { // get the handle
        ObjectHandleData* data = vt_amb->retreiveData(desc->objectHandle);
        if (data) {
            entity = data->ent;
            if (entity) {
                store(entity); // associate handle with entity
                local = (hlaEmitterBeam*) entity->findComponent("hlaEmitterBeam");
                if (local) {
                    *local = rti_data;
                    local->init();
                }
            }
        }
    }
}

// local is set automatically at update because entity is
// automatically retrieved from the handle.

if (local) {
    local->draw();
}
```

`vt_amb` is a pointer to the `vtAmbassador` class.

The `getDesc()` function with an Object name returns the `ObjInstDesc` that contains the RTI Handle for this named Object.

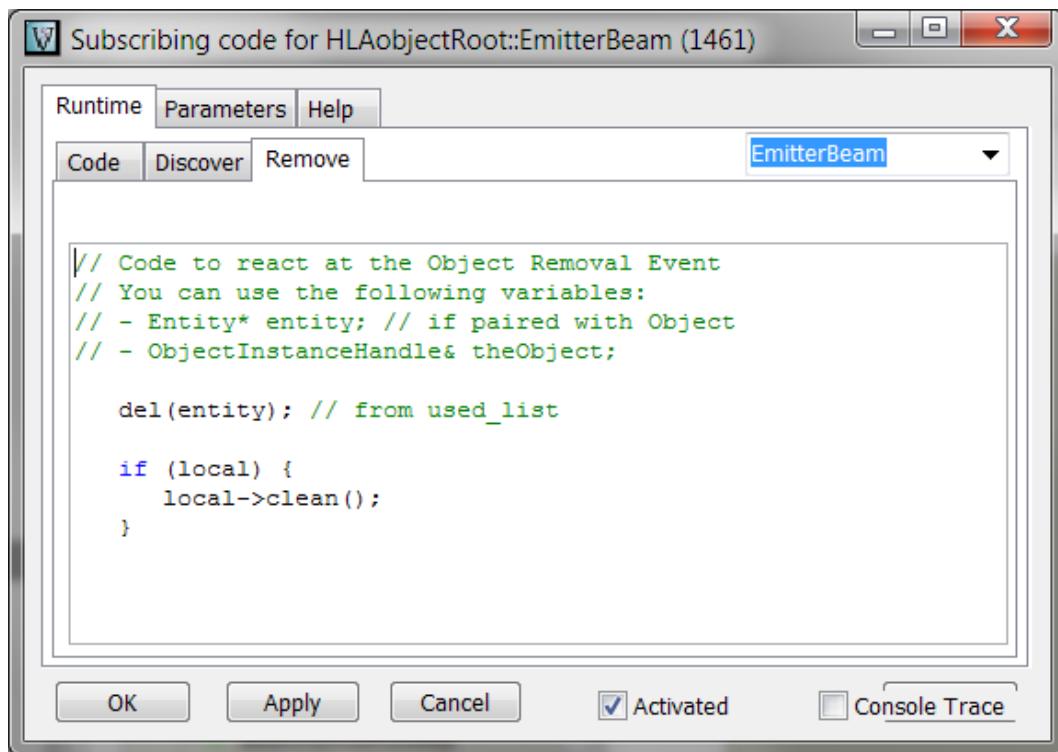
The function `retreiveData()` with an Handle returns the tuple (`Handle`, `Entity`, `UserData`).

With these two functions, we are able to associate the local data pointer with the Handle and make it point to the `hlaEmitterBeam` component.

Later coming updates will automatically set the data into the `DataModel` object of the Entity, pointed to by `local`.

The `EmitterBeam` main code will then just draw the beam on the map (`local->draw()`)

The same way, we must not forget to clean the `DataModel` drawing by calling `clean()` in the Remove panel of the `EmitterBeam` Object :



## • hlaEmitterBeam DataModel

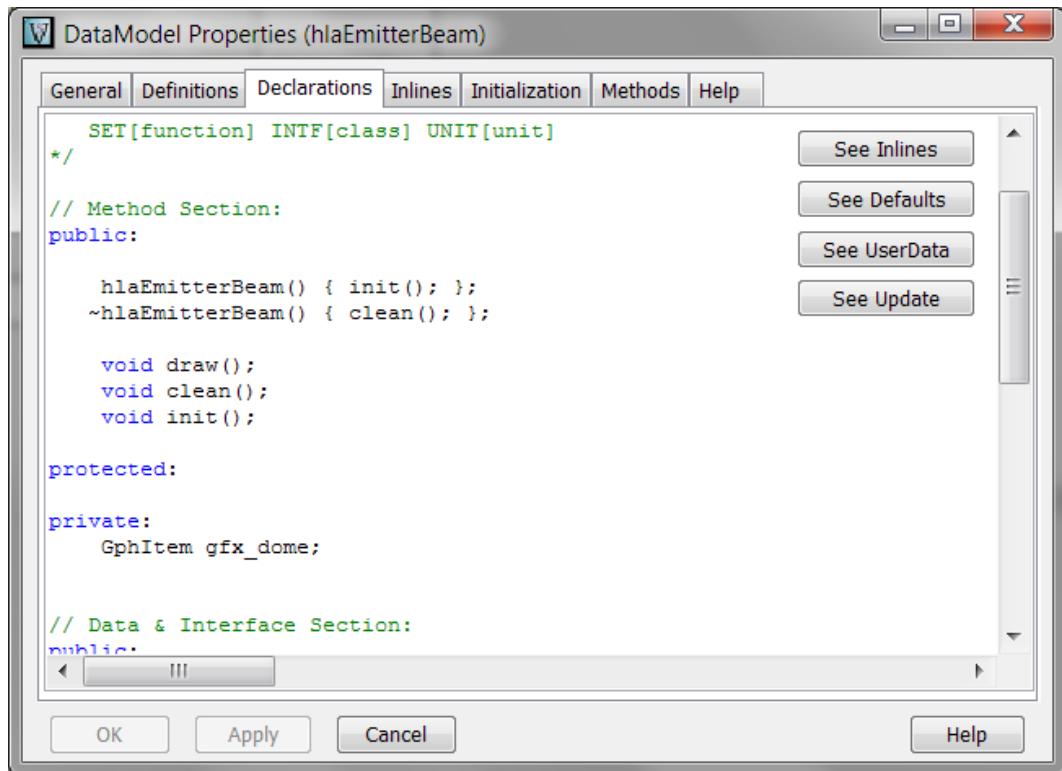
Once the DataModel have been constructed from OMT definitions, they can be augmented with user data and methods.

To prevent later overwriting of these DataModels, it is a good idea to lock them to secure the changes.

Here, we want the [hlaEmitterBeam](#) DataModel to be able to draw something according to its values.

We will use a [Gph\\_Item](#) and three functions: `init()`, `draw()` and `clean()`:

## RPR-FOM



The Methods panel will get the declarations of the three functions:

```
// *****
void Dml::init()
{
    gfx_dome.id = -1;
}

// *****
void Dml::draw()
{
    if (gfx_dome.id <0) { // not displayed yet
        // Dome
        vt_rtc->gui_map.drawDome(entity->getName(), // base
                                    BeamAzimuthCenter-BeamAzimuthSweep,
                                    BeamAzimuthCenter+BeamAzimuthSweep,
                                    0, 0,
                                    EffectiveRadiatedPower*10,
                                    entity->db->getColor());
        gfx_dome = vt_rtc->gui_map.getLastItem();
    }
    else {
        if (gfx_dome.id >= 0) {
```

```

    gfx_dome.color = entity->db->getColor();
    gfx_dome.dome.min_azim = BeamAzimuthCenter-BeamAzimuthSweep;
    gfx_dome.dome.max_azim = BeamAzimuthCenter+BeamAzimuthSweep;
    gfx_dome.dome.min_elev = DEG2RAD(0);
    gfx_dome.dome.max_elev = DEG2RAD(0);
    gfx_dome.dome.radius = EffectiveRadiatedPower*10;
    vt_rtc->gui_map.updateGraphic(gfx_dome);
}
}

// *****
void Dml::clean()
{
    if (gfx_dome.id > -1) {
        vt_rtc->gui_map.removeGraphic(gfx_dome.id);
        gfx_dome.id = -1;
    }
}

```

## • Interactions

By pressing the 'b' button in the F18.exe window, the closest entity from the F18 will receive a [WeaponFire](#) interaction and 3 seconds later a [MunitionDetonation](#). We will subscribe to these two Interactions.

**WeaponFire:** We want to draw a blue circle around the shooter then a line between the shooter and the target. For that, we need to get the following three parameters: [FiringLocation](#), [FiringObjectIdentifier](#) and [TargetObjectIdentifier](#)

[FiringLocation](#) Attribute code to draw the blue circle:

```

rti_data.FiringLocation <- (WorldLocationStruct);

ENDIAN_SWAP_DOUBLE64(rti_data.FiringLocation.X);
ENDIAN_SWAP_DOUBLE64(rti_data.FiringLocation.Y);
ENDIAN_SWAP_DOUBLE64(rti_data.FiringLocation.Z);

WCoord pos(WC_ECEF, rti_data.FiringLocation.X,
rti_data.FiringLocation.Y, rti_data.FiringLocation.Z);
pos.convertECEFtoXYZ();
R:>gui_map.drawCircle(pos, 100, clBlue, 3);

```

## RPR-FOM

Then, from the [WeaponFire](#) Object code, we draw the line:

```
Vt_Entity* from = S:findEntity(rti_data.FiringObjectIdentifier.ID);
Vt_Entity* to = S:findEntity(rti_data.TargetObjectIdentifier.ID);

if (from && to) R:gui_map.drawLine(from->pos, to->pos, clRed, 2);
```

**MunitionDetonation:** We will use the [TargetObjectIdentifier](#) to kill the Entity and use the [DetonationLocation](#) to draw a red circle of 3 seconds:

```
rti_data.DetonationLocation <- (WorldLocationStruct);

ENDIAN_SWAP_DOUBLE64(rti_data.DetonationLocation.X);
ENDIAN_SWAP_DOUBLE64(rti_data.DetonationLocation.Y);
ENDIAN_SWAP_DOUBLE64(rti_data.DetonationLocation.Z);

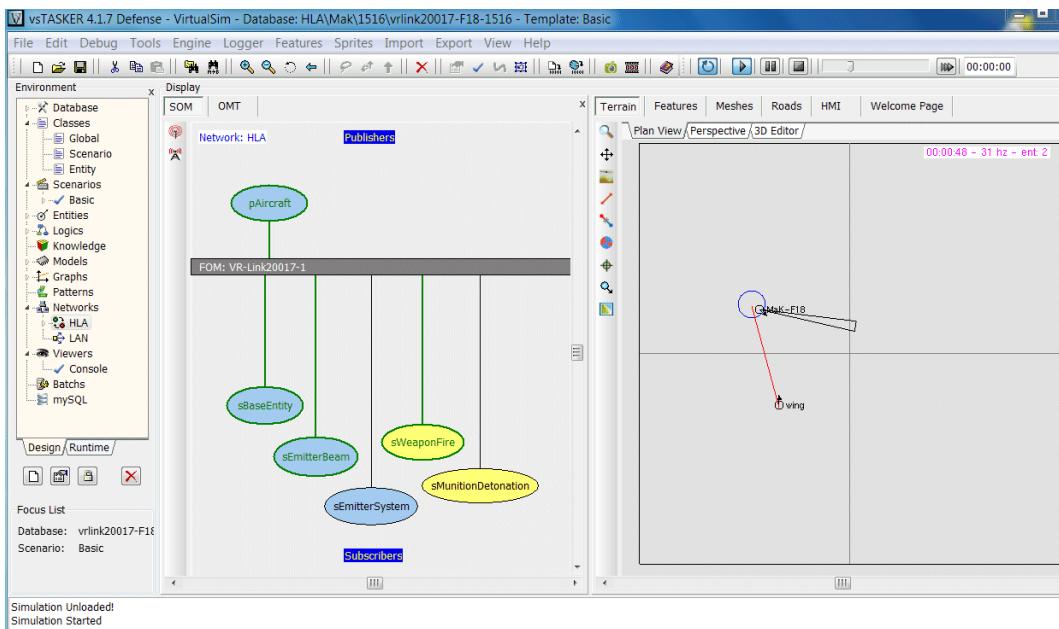
WCoord pos(WC_ECEF, rti_data.DetonationLocation.X,
rti_data.DetonationLocation.Y, rti_data.DetonationLocation.Z);
pos.convertECEFtoXYZ();

R:gui_map.drawCircle(pos, 100, clRed, 3);
```

[MunitionDetonation](#) Object code:

```
Entity* target = (Entity*)
S:findEntity(rti_data.TargetObjectIdentifier.ID);
if (target) {
    target->status->setDamage(_Damaged);
    target->dyn->startCrash();
}
```

- **Running the Demo**



*This demo is available in HLA/Mak/1516/vrlink20017-F18-1516 and Hla/Mak/1516/vrlink-F18-1516.*



*See the Scenario description to know how to launch the VR-Link demo part.*

*If you do not have VR-Link, you can still use the /HLA/Mak/1516/multisensors\_master1516 and slave with two vsTASKER running (available also on Pitch/1516)*

*Rely on the Scenario explanations to see how to manipulate the demos.*

## Importing SOM

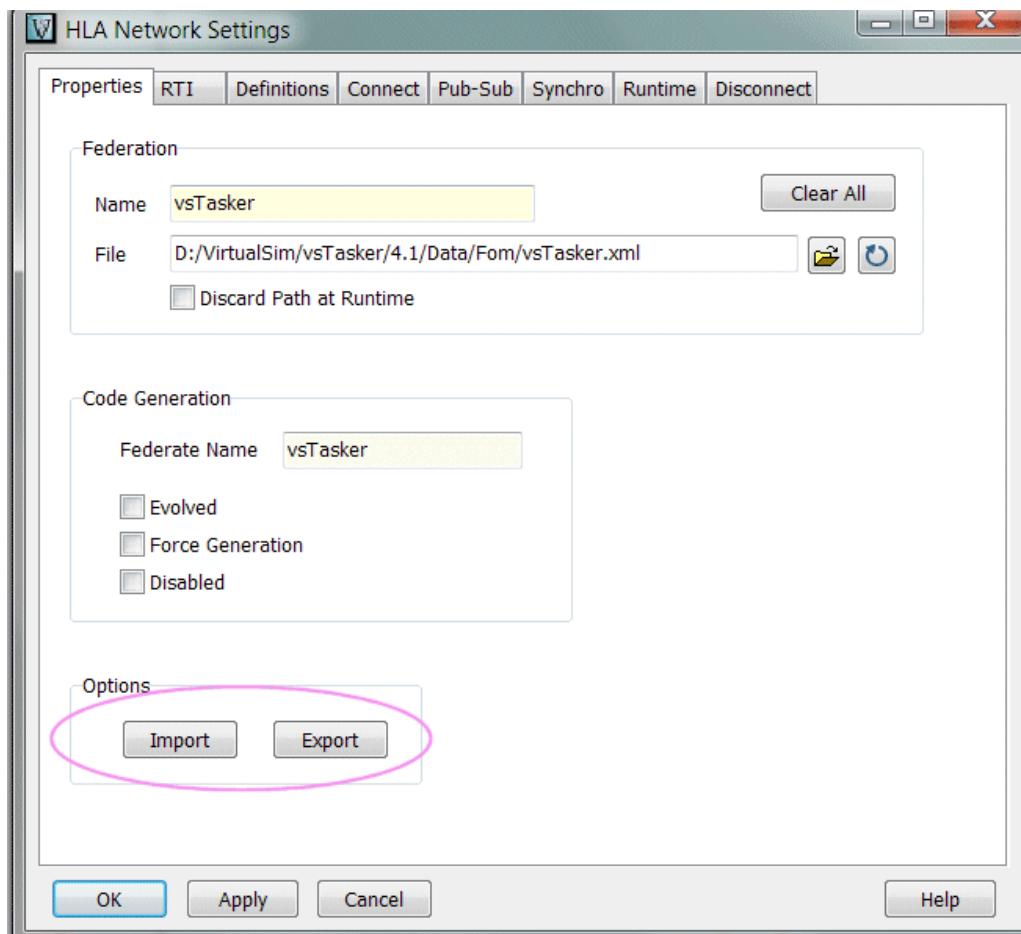
# Importing SOM

vsTASKER provides some built-in SOM user can directly import to start going HLA without too much trouble.

The SOM Importer gather the OMT and all the FedItems, but does not import the DataModels.

For this reason, the DataModels must be generated based on the OMT definition (which depends on the FDD file)

To Export or Import a current SOM definition, use the following buttons:



## **CIGI**

The chapter will introduce the use of CIGI under vsTASKER.



*Specific license is mandatory. Contact vendor if you need this module.*

## Concepts

# Concepts

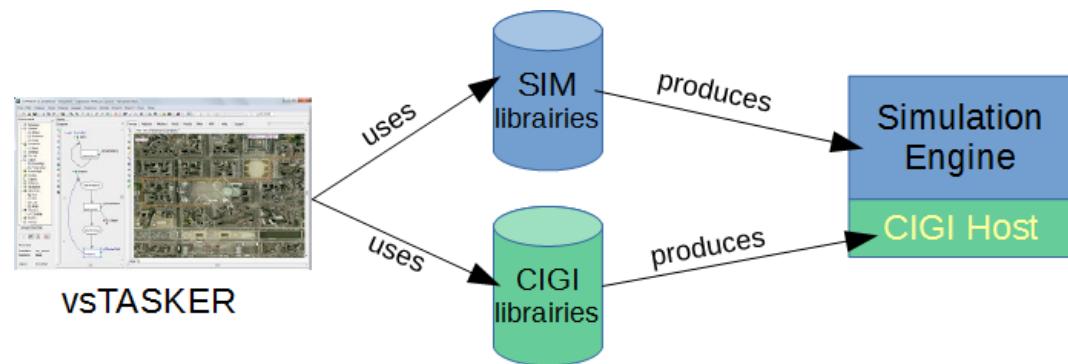
CIGI is a data packaging protocol and thus does not depend upon a specific physical communications medium or transport protocol. Any suitable physical medium may be used, including Ethernet, Token Ring, optical fiber, shared memory, etc. The transport protocol(s) used should depend upon performance and what is appropriate to the communications hardware. This document assumes the use of the User Datagram Protocol (UDP) over Ethernet for ease of discussion.

CIGI data exchanges is based on standardized data packets.

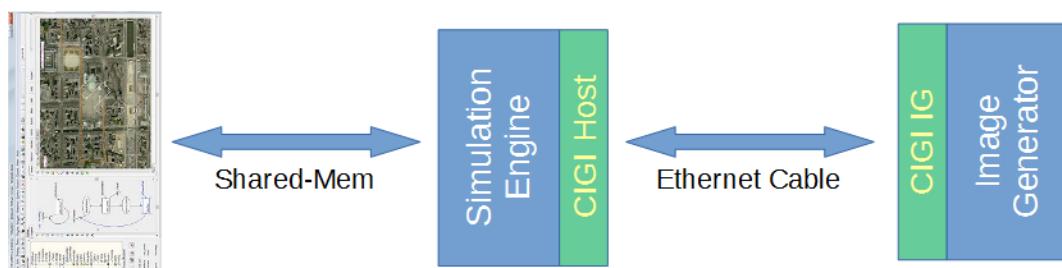
A complete documentation can be found here: [https://www.sisostds.org/stdsdev/tracking\\_final/doc\\_191/siso-std-013-draft\\_cigi\\_version4.pdf](https://www.sisostds.org/stdsdev/tracking_final/doc_191/siso-std-013-draft_cigi_version4.pdf)

The way vsTASKER uses CIGI is by code generating all calls to the CIGI libraries from definitions and messages created into the database.

It then produces the CIGI host side which will be included into the simulation engine to produce a unique and deployable executable.



Simulation runtime will then take this configuration:



## • CCL

vsTASKER is using CIGI CCL (<http://cigi.sourceforge.net>) library under GNU license. Include files are located in `/include/CIGI` and libraries per compiler in `/Lib/vc100/CIGI` (for [VisualStudio 2010](#)):

1. **ccl.lib** and **cclD.lib**
2. **networking.lib** and **networkingD.lib**

User can replace these libraries by versions of his own if he extends the messages or add new fields in existing ones.



*Some vendor specific messages have been added to the library (for VBS). Source code of such messages can be requested to VirtualSim free of charge for licensed users.*

## Definition Files

# Definition Files

CIGI messages is using ID to select terrains, models or articulated parts.  
In order to have a handy way to setup a CIGI environment, definition files will be used.

By default, some files have been located in [/Runtime/Settings](#)

## • **terrains.def**

This file is used in **CIGI property** window to specify which terrain must be loaded at start.

The content is as follow:

```
1=pendleton:/img/pendleton.jpg
2=my_terrain
3=...
```

### grammar:

*id=name{:*image*}*

//: any comment, line not processed

*id*: number the IG does associate with the terrain

*name*: what to display for the user to represent the terrain

*image*: optional, small image (500x500) located at the same level as the **def** file. If inside a folder, mention it (ie: [/img/whatever.jpg](#))

## • **entities.def**

Used by component **CigiEntity**

The content is as follow:

```
// entities
#afg
#civilian
13443=afg_civ_gaz_24_black_x:img/preview_gaz_24_black_ca.jpg
13444=afg_civ_gaz_24_blue_x:img/preview_gaz_24_blue_ca.jpg
>-- // civilian
>-- // afg
```

**grammar:**

*id=name{:image}*

//: any comment, line not processed

*id*: number the IG does associate with the 3D model

*name*: what to display for the user to represent the model

*image*: optional, small image (500x500) located at the same level as the def file. If inside a folder, mention it (ie: /img/whatever.jpg)

# *name*: create a branch with this name. All successive entries will belong to this branch

#--: close the current branch

- **lifeforms.def**

Used by component **CigiLifeform** (which inherits from **CigiEntity**)

- **weapons.def**

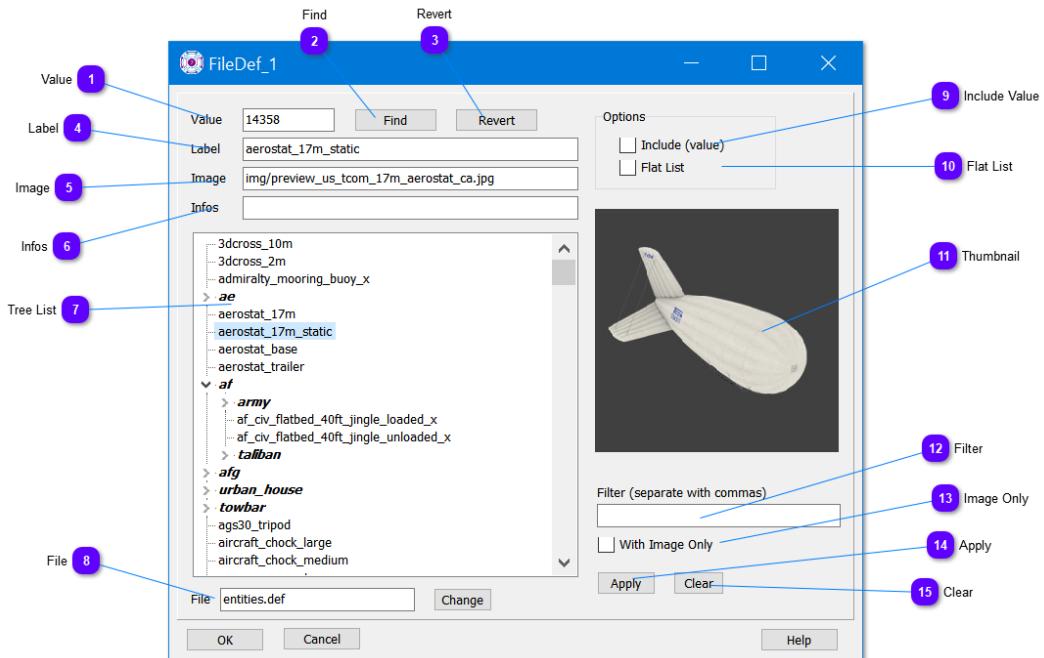
Used by component **CigiWeapon** (which inherits from **CigiEntity**)

- **munitions.def**

Used by component **CigiMunition** (which inherits from **CigiEntity**)

## Property Window

# Property Window



### 1 Value

Value

ID value extracted from the line:

*id = label {; image file} { // comment }*

### 2 Find

Find in the list the entry whose ID is set in the field. You can change it and press Find to retrieve it and fill all fields + image if defined

### 3 Revert

Revert to the default value mentioned in the field from which this window has been opened.

**4 Label**

Label aerostat\_17m\_static

Label value extracted from the line:  
`id = label { : image file } { // comment }`

**5 Image**

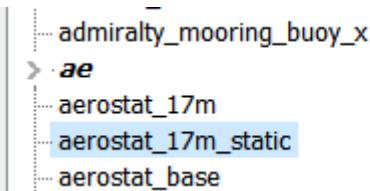
Image img/preview\_us\_tcom\_17m\_aerostat\_ca.jpg

Optional Image file extracted from the line:  
`id = label { : image file } { // comment }`  
 It must be separated from the label with a colon.

**6 Infos**

Infos

Optional comment extracted from the line:  
`id = label { : image file } { // comment }`  
 It must be located at the end of the line, prefixed with //

**7 Tree List**

Description of all the entries from the definition file.  
 Hierarchies are defined using `# label` and `#--`

**8 File**

File entities.def

Change

Name of opened file. Use **Change** button to select another one.

## Property Window

9

### Include Value

 **Include (value)**

If checked, the tree list will also display the ID between parenthesis after the label.

**Apply** button must be depressed to rebuild the tree list.

10

### Flat List

 **Flat List**

If checked, the tree list will display all entries disregarding the hierarchy definitions.

**Apply** button must be depressed to rebuild the tree list.

11

### Thumbnail



Display here the preview image (thumbnail) defined in the line, for the selected entry.

The image should be located in the level below the loaded definition file.

12

### Filter

Filter (separate with commas)

Enter here any motif to look for in the labels.

**Apply** button must be depressed to rebuild the tree list.

13

### Image Only

 **With Image Only**

If this option is checked, only the entries with image will be kept.

**Apply** button must be depressed to rebuild the tree list.

**14 Apply**

 Apply

Rebuild the tree list according to the selected options.

**15 Clear**

 Clear

Set the options to default values.

# First Steps

Although you could open a CIGI example and modify it, we will see here how to convert an existing database to enable the CIGI capability.

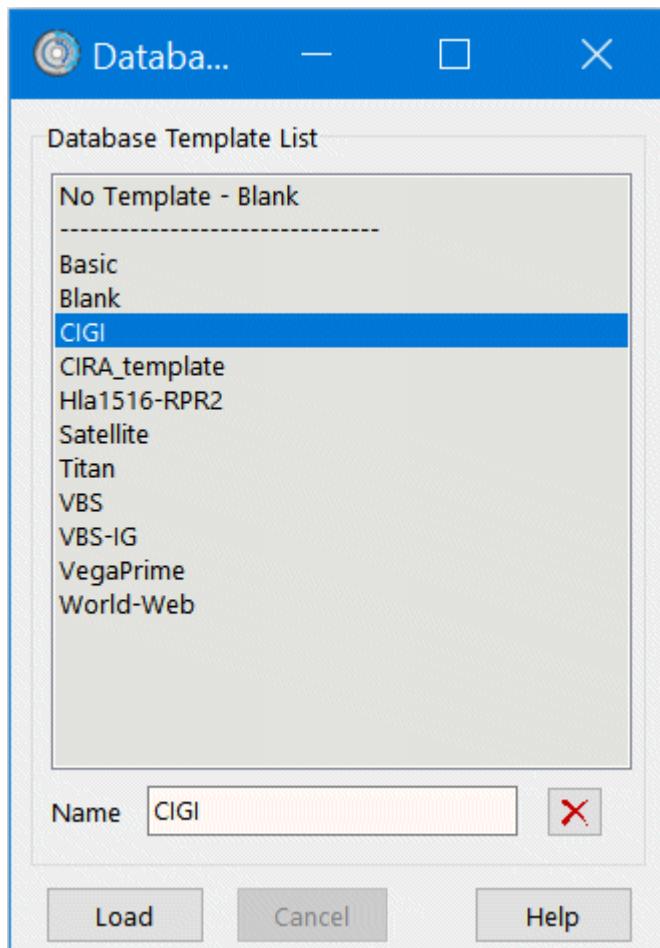
## • Using the Template

To create a new simulation using CIGI (and not making a current one CIGI compatible), the simplest way is to use a template.

Do the following:

**File::New**

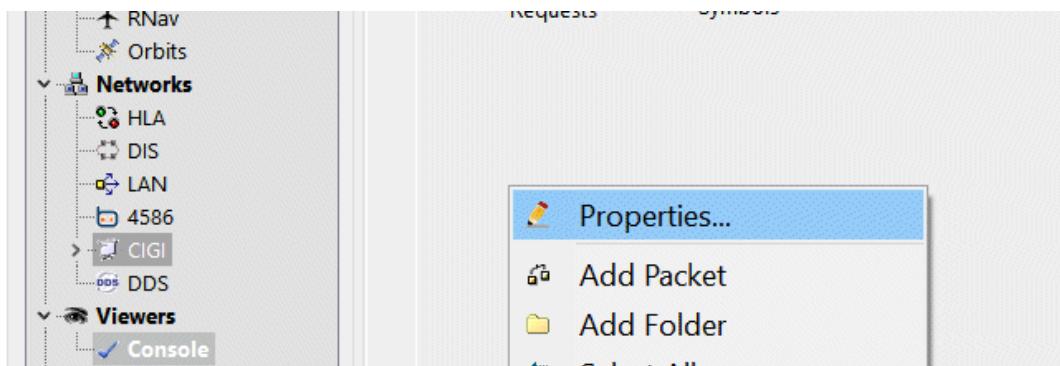
Then select CIGI template:



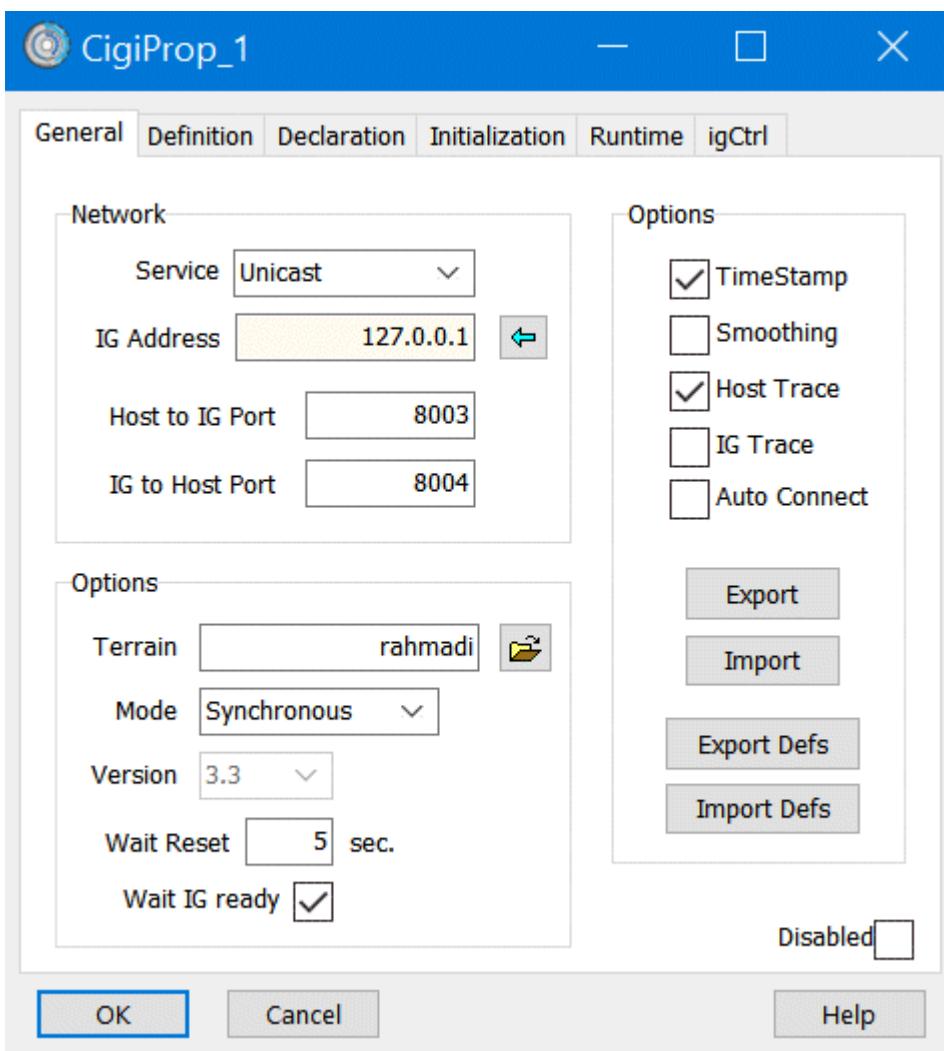
Save then the database under the name you want (ie: [test\\_cigi](#))

Now, let's check the CIGI configuration.

Go to **Network::CIGI** and right click to select Properties:



You will get this window:



Select the network your IG is using. Refer to its documentation.

## First Steps

The terrain to use is selected from a list defined in a `.def` file. Refer to this chapter for [definition files](#).

For ie, open the `terrains.def` file in [/Runtime/CIGI/Settings/OSG](#), then select the one to be used with CIGI. vsTASKER will send the initialization message to the IG with the corresponding ID.

You can (and must) define your own `terrains.def` file with the list of pairs to display in the drop down menu.

First thing is to check that so far, the Host and IG can connect.

Recompile and run the SIM. You should obtain this output:

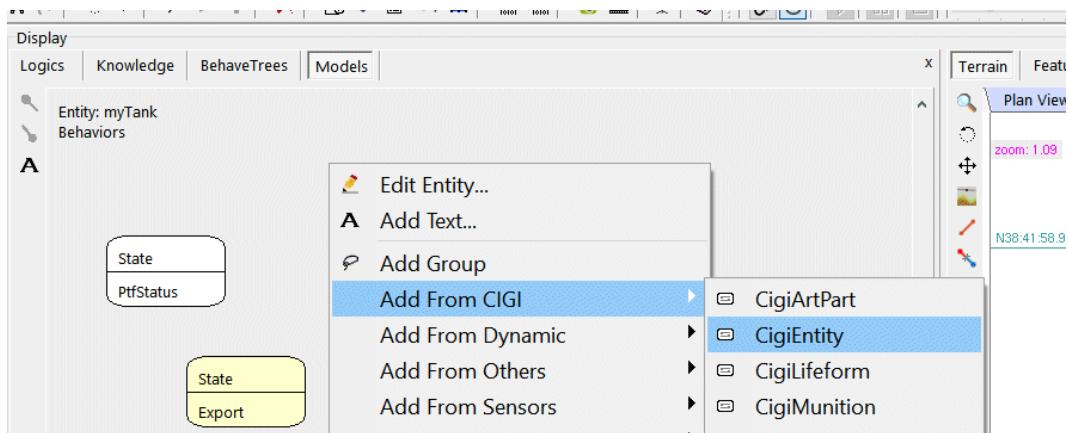
```
Having loaded environnement for: test_cigi
Initialize Shared-Mem...
Setting up Networks...
Initialize Networks...
Initializing ports to CIGI:
127.0.0.1: Host -> 8003 -> IG -> 8004 -> Host
Successfully connected to CIGI IG server
Force reset IG (Synchronous), load database 1
Waiting for IG Reset...ok
Waiting for IG Operate...ok
Initialize Logger...
Initialize HMI manager...
Initialize SQL...
Initialize Global...
RTC initialized successfully
Simulation Engine is ready
```



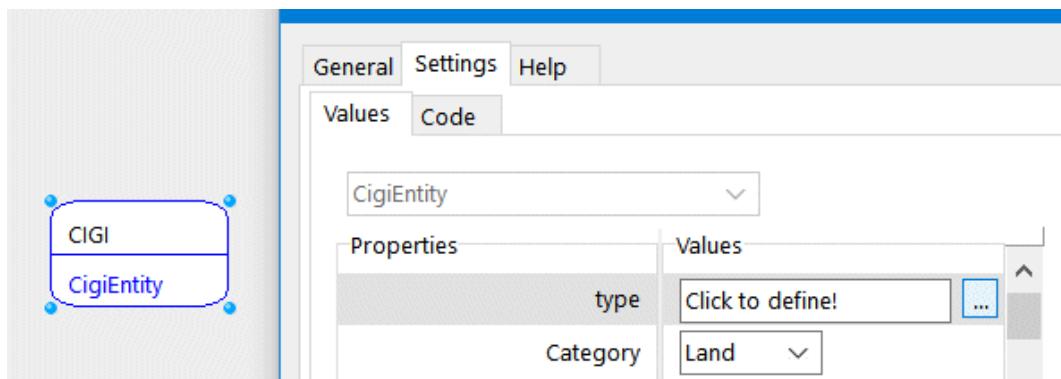
You can use VisualStudio to compile the simulation engine. Use the proper solution in [/Runtime/CIGI](#) and include the `test_cigi_code.cpp` and `test_cigi_intf.cpp` files from [/Gen](#) (if you used `test_cigi` to name the database of this example).

Now, open the terrain corresponding to the one in the IG (or set the same coordinates in vsTASKER) and add a Tank. Drop on the map a default entity then name it `myTank`.

Select the entity (`myTank`) then select **Models** and right-click to import the **CigiEntity** component:



Double click the **CigiEntity** imported component and set the model type:



The entity types are defined into a specific file which pairs the id with the name. Refer to this chapter for [definition files](#). Click on ... then open the **entities.def** file which does that.

You can (and must) define your own **entities.def** file with the list of (id=name) pairs to fill the drop down menu.

The IG must have the same ID for the selected 3D model type. You have to insure the matching is correct.

Once this is done, recompile and run.

The entity **myTank** should be created in the IG with the 3D model you choose. Select it on the map and the camera should move and focus on it.

### • Enabling CIGI on an existing database

If you want to make an existing non-CIGI simulation database able to use the CIGI protocol, you must follow the below procedure:

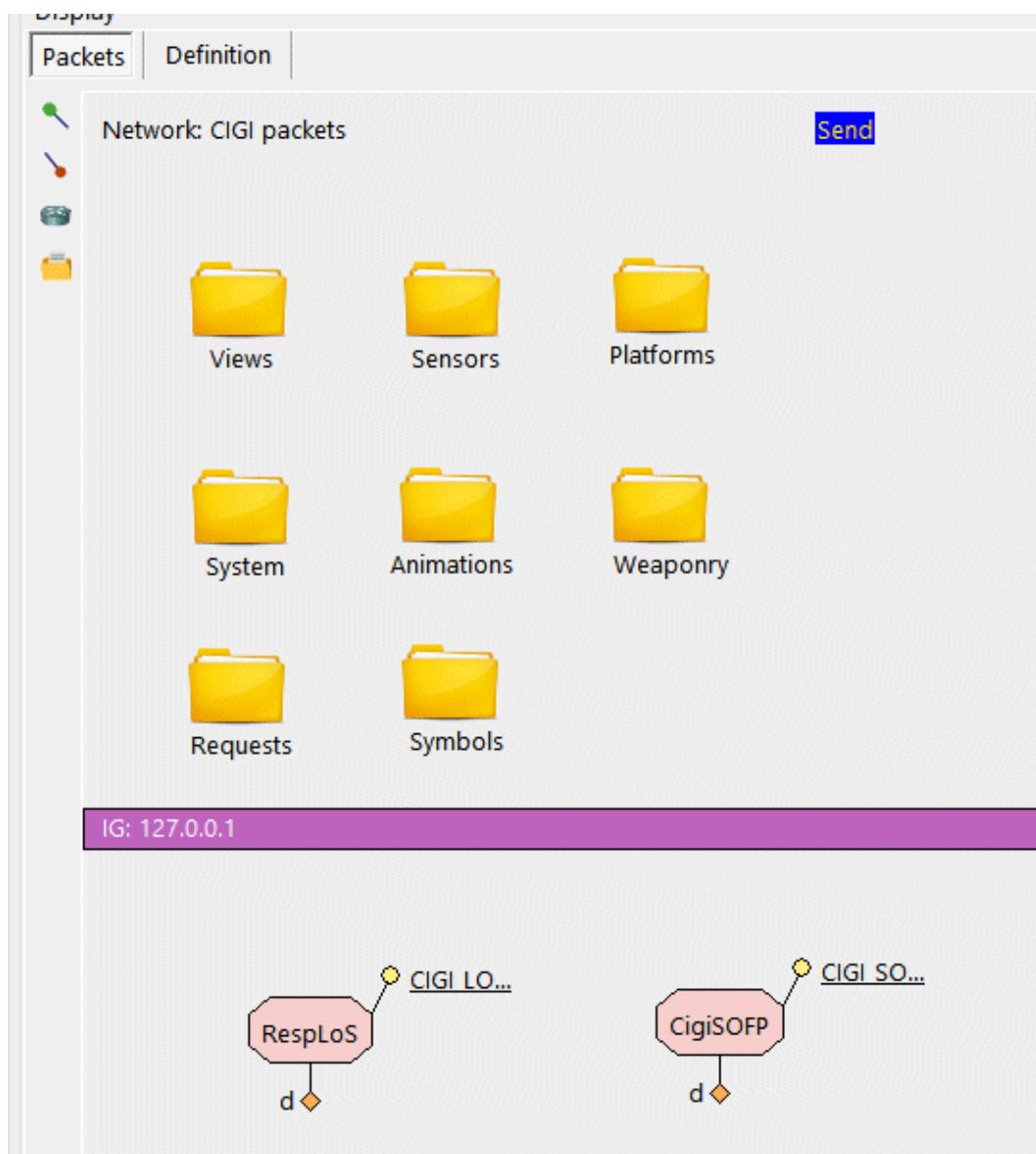
## First Steps

1- Open the database

2- Select **Networks::CIGI**, then right click for **Properties** (or double click the **Display** background)

3- On the CIGI Property window, click **Import**, then in the folder **CIGI**, select **default** (or **Vbs-ig** if you want to use this IG)

The Display panel should be as below:

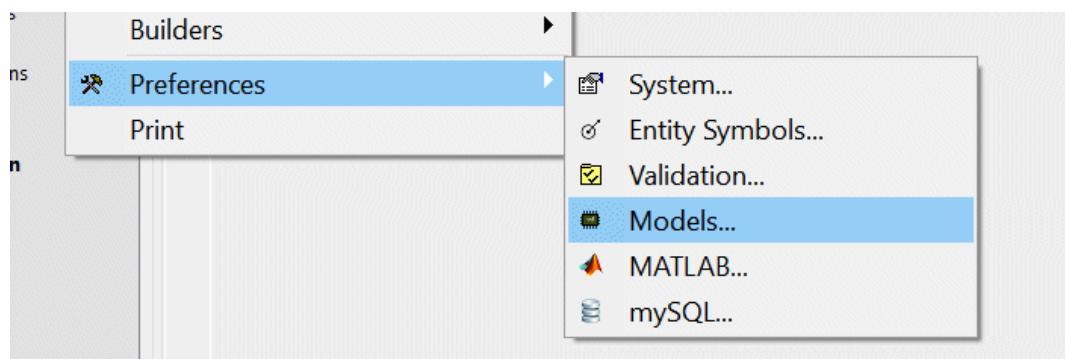


4- Open again the CIGI Property window and set the following:

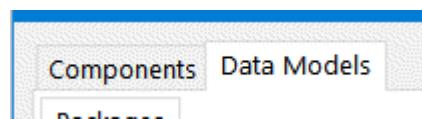
- IP address for the network
- Ports

- Terrain ID

**5-** We may need to add the **CIGI** Model package to vsTASKER is not already added.  
Select menu **Tools::Preferences::Models**



On the **Model Configuration** window, select **Data Model** tab



If **Cigi** is not visible in the list, click on the open file icon then select file:  
**cigi\_dm.lst** (should be in **/Models**) and click on   
The window should now list all packages + CIGI



**6- Select Environment::Model**



and in the **Display** area, right click to **Add** a new **Container**. Name it **CIGI**.

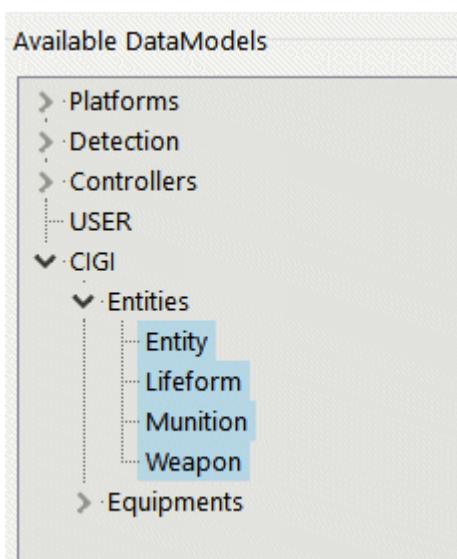
## First Steps



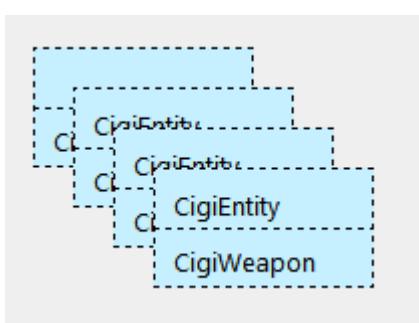
Now double click the icon to open it.

On the Display background, right click and select **Add Data Model**

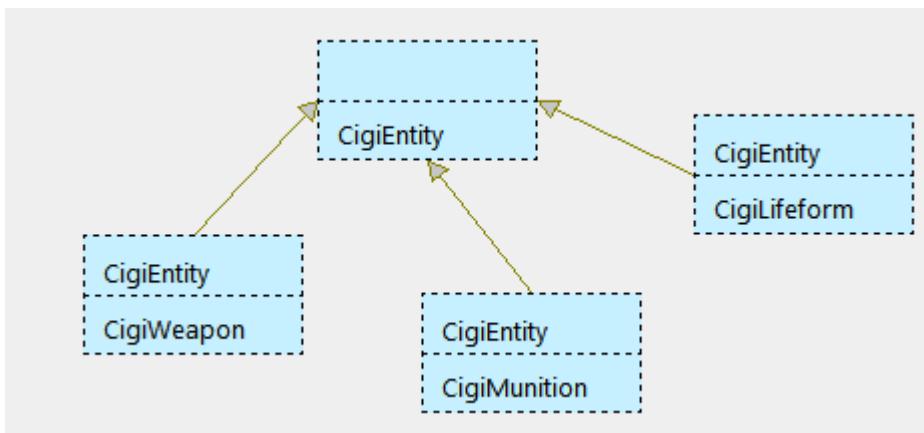
In the **Importer** window, expand the CIGI list (if you do not have it, go back to 5) and select **Entity**, **Lifeform**, **Munition** and **Weapon** Data Models (use *Ctrl* key for multiple selection)



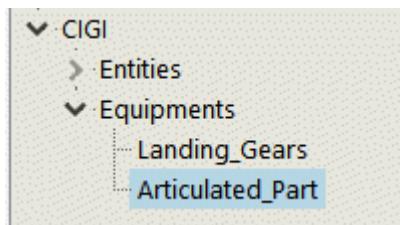
Then **Import** button



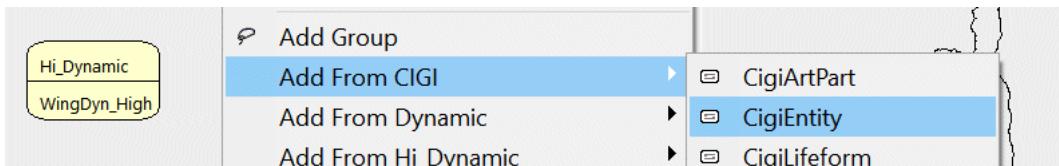
You can reorganize the icons



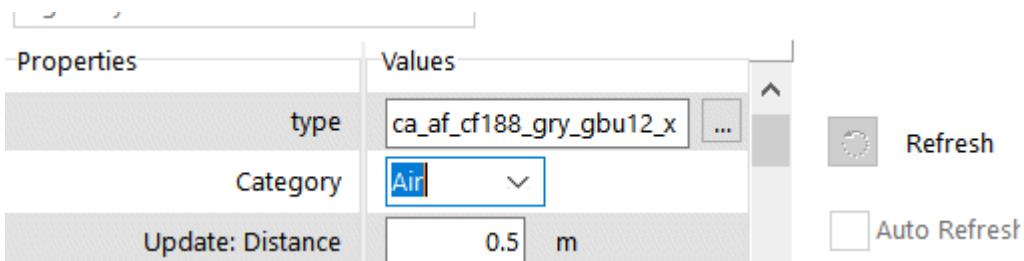
After that, import the [Articulated Part Data Model](#)



**7-** Now select your entities and in their Models panel, add the [CigiEntity](#) component:



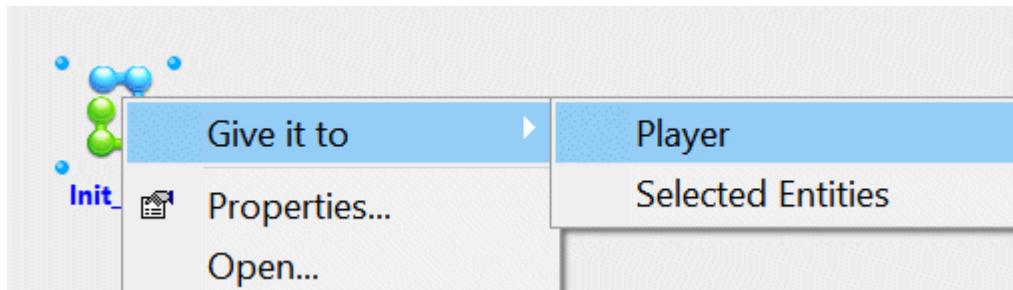
Open (double click) the Component Model and set the 3D model based on a configuration file according to your IG:



**8-** Finally, import the [Cigi\\_Init](#) logic. Select **Environment::Logic**, then right click and **Import ...**

In the File selection dialog, open **CIGI** (in /Shared) and select **Init\_CIGI.lgk**  
Click on the logic icon, right click and give it to **Player**

## First Steps



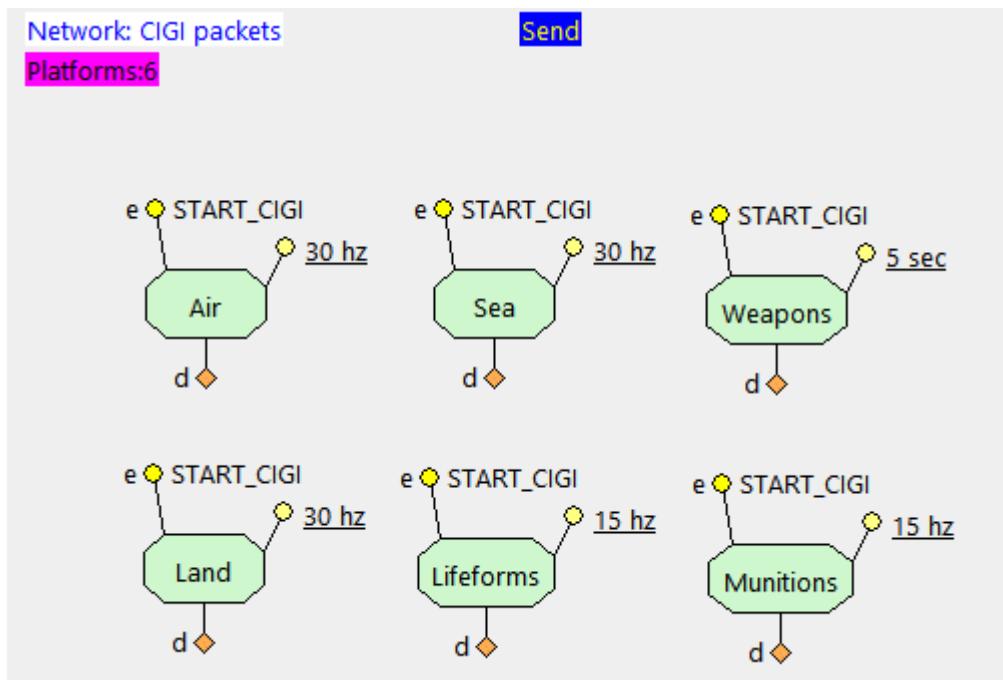
**9-** Recompile and run the simulation.

You should get the following output console. Make sure that you have *ok* for *IG Reset and Operate*:

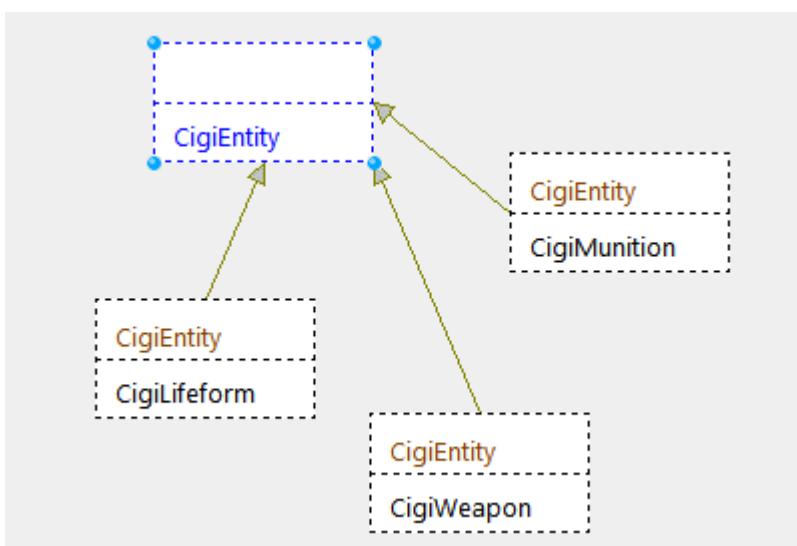
```
Initialize Shared-Mem...
Setting up Networks...
Initialize Networks...
Initializing ports to CIGI:
225.0.0.181: Host -> 8003 -> IG -> 8004 -> Host
Successfully connected to CIGI IG server
Force reset IG (Synchronous), load database 1
Waiting for IG Reset...ok
Waiting for IG Operate...ok
Initialize Logger...
Initialize HMI manager...
Initialize SQL...
Initialize Global...
RTC initialized successfully
Simulation Engine is ready
```

# Messages

The way vsTASKER handles CIGI is by encapsulating a message into an [Packet](#) object (which can also be seen as a class) with all latitude for any usage. The thing to remember is that any CIGI [Packet](#) defined in the GUI will generate a class with the same name (or like) and only one instance.



In the above definition of [Packets](#) (*Air*, *Sea*, *Land*...), we have separate class definitions which all are using the [CigiEntityCtrlV3\\_3](#) class of the CCL library (in fact, they are all using the [CigiEntity](#) Data Model which used the [CigiEntityCtrlV3\\_3](#) class):



## Messages

Each Packet is running at a given frequency to ensure update of the IG. The more update, the better the output but the more load of the network. When many entities are in stake, it is advisable to manage wisely the update rate.

As each Packet class generates one instance, it can be easily accessed from any place in the code.

For example, the Air Packet above will be accessed this way:

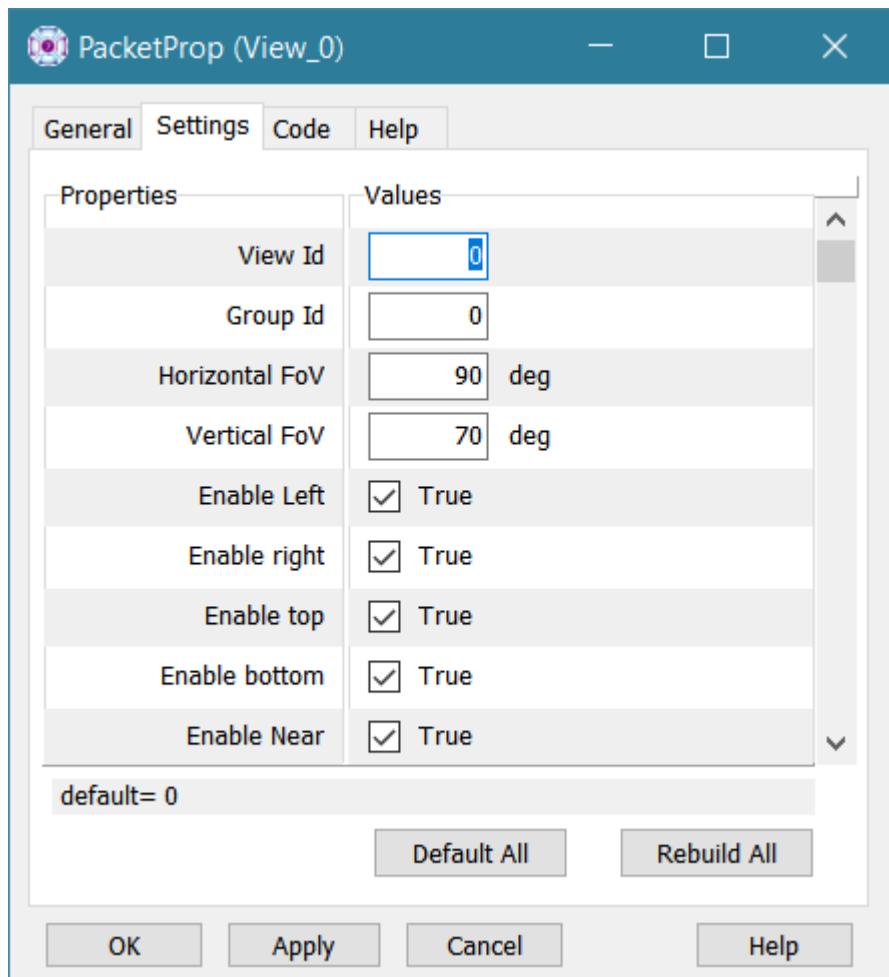
```
cigi.Air->...
```

If the user has added the method: `update ()` in the Packet definition, the method will be accessible from the code:

```
cigi.Air->update();
```

Some Packet need to handle several entities. This is the way above (Air, Sea...) Packets are doing. They have a list of `CigiEntity` objects to handle and they perform the update when necessarily.

`CigiViewDefV3` message, can be called on event (or from the code) and issue only one message. For such Packet associated with only one message, the message parameters can be mapped with local variables for user to setup:



Of course, it would make no sense to allow a static data interface for a Packet with multiple instances inside. Reason why such Packets uses an array of Data Models attached to each entity with their own specific data interface.

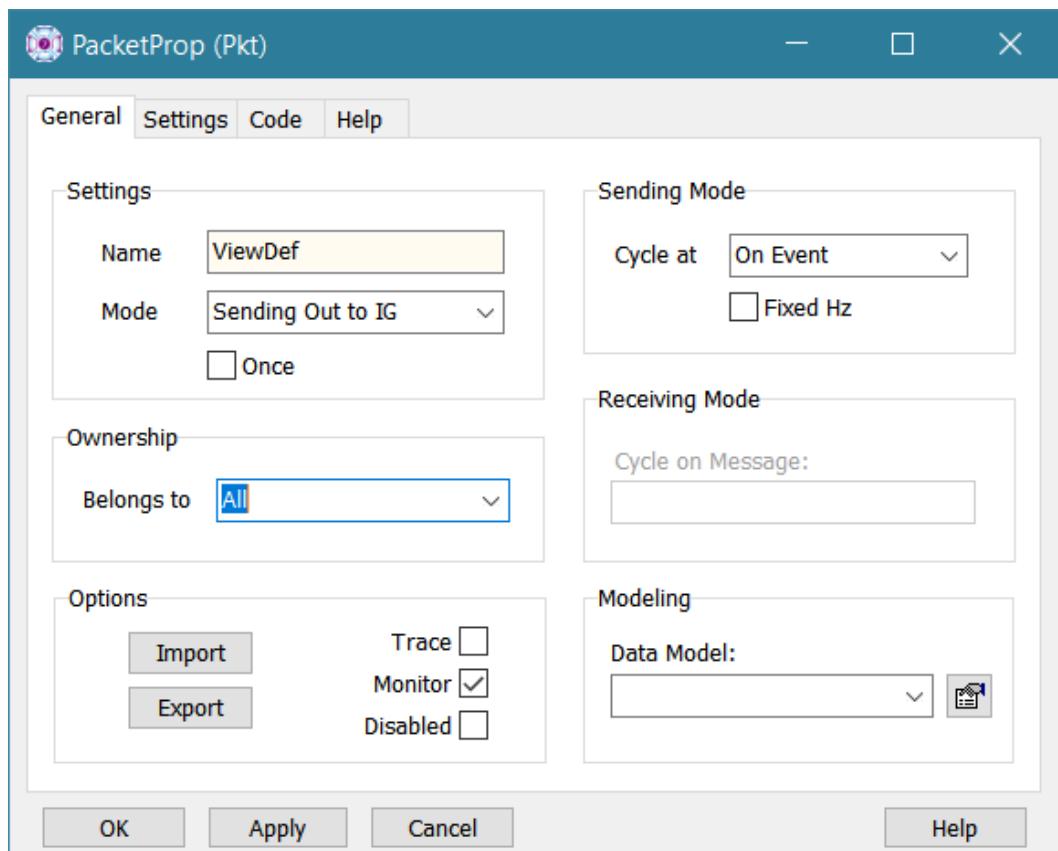
## Host to IG

# Host to IG

The upper section of the CIGI Packet definition is reserved for Sending messages from host (vsTASKER simulation engine) to the IG.

We will create and explain the View Definition message. All other messages will follow the same principle.

First, let's create the Packet in the upper section and set it up like below:



Now, in the **Code::Declaration**, let's add the following code:

```
CigiViewDefV3 def;
```

In order to allow the user to set the values of some variables of the `def` object, we can define them specifically for the dynamic data interface.

To simplify, we will just let the user specify the **view id** and the **fields of view**. All the other values will be defaulted by the **Packet**.

Below the declaration above, add the following code:

```
public:  
    int  view_id;  //&& DEF[0]
```

```
float h_fov; //&& LBL[Horizontal FoV] DEF[90] UNIT[deg]
float v_fov; //&& LBL[Vertical FoV] DEF[70] UNIT[deg]
```

Now, below, let's add two methods. One for setting the view and the other one for updating the `def` class with user and default data.

```
public:
    void setCurrentView();
    void changeView(int new_id, float h, float v);

private:
    void setupParams();
```

In the **Initialization** panel, we call the `setupParams` function:

```
case RESET: {
    setupParams();
} break;
```

In the **Methods** panel, we define our two declared functions (explanations in the code below):

```
void Pkt::setCurrentView()
{
    CGI:view_id = this->view_id; // store globally (at the CIGI settings level)
    the selected view
    tic(); // send the message
}

void Pkt::changeView(int new_id, float h, float v)
{
    view_id = new_id; // replace the current id
    h_fov = h; // replace the horizontal fov
    v_fov = v; // replace the vertical fov
    setupParams(); // send the data to the def object
    setCurrentView(); // send the object to the IG
}

void Pkt::setupParams() // send the local user data to the def object
{
    def.SetViewID(view_id);
    def.SetGroupID(0);

    def.SetFOVLeftEn(true);
    def.SetFOVRightEn(true);
```

## Host to IG

```
def.SetFOVTopEn(true);
def.SetFOVBottomEn(true);

def.SetFOVLeft(-h_fov/2);
def.SetFOVRight(+h_fov/2);
def.SetFOVTop(+v_fov/2);
def.SetFOVBottom(-v_fov/2);

def.SetFOVNearEn(true);
def.SetFOVFarEn(true);
def.SetFOVNear(0);
def.SetFOVFar(10000);

def.SetProjectionType(def.ProjectionTypeGrp::Perspective);
}
```

Now, in the **runtime** (part called at the frequency specified in the Packet settings - here, manually or [OnEvent](#)), let's send the message:

```
*mgr->0msgPtr << def;
mgr->nb_pkt++;

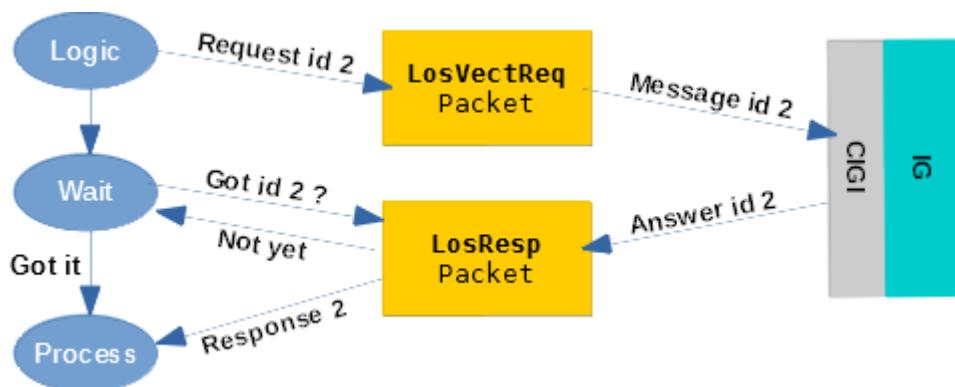
return DONE;
```

## IG to Host

The lower section of the CIGI Packet definition is reserved for **receiving** messages from IG to the host (vsTASKER simulation engine).

Because of the asynchronous communication between Host and IG, each response is tagged with the same id as the request.

The principle retained to treat asynchronous response to request, without blocking the single thread of the simulation engine, is to give to a logic the task to talk to the IG.



Sending the Line of Sight request message will be done on the same principle as here. Instead of the `CigiViewDefV3`, we will use the `CigiLosVectReqV3_2` class. In the `LoVectorReq` Packet, the code will be the following:

```

void Pkt::request(WCoord from, float az, float el, int id)
{
    from.convertToLLA();
    if (az > 180) az -= 360; // [0..359] -> [-179..180]

    req.SetLosID(id);
    req.SetSrcCoordSys(CigiBaseLosVectReq::CoordSysGrp::Geodetic);
    req.SetVectAz(az);
    req.SetVectEl(el);
    req.SetSrcLat(from.lat);
    req.SetSrcLon(from.lon);
    req.SetSrcAlt(from.alt);
    tic();
}

```

In the **Logic** object, the code is:

## **IG to Host**

```
cigi.LosVectReq->request(E:pos, E: dyn->getHeading(), E: dyn->getElevation(), 2);
```

In the **Wait** object, the code is:

```
if (!cigi.LosResp->getResponse(2, false)) // continue asking
```

In the **Process** object, the code is:

```
CigiLosRespV3_2* response = cigi.LosResp->getResponse(2, true) //  
true = remove it from the list  
// process response  
...  
delete response;
```

## Testing with MPV

To test vsTASKER CIGI, we can use the Multi-Purpose Viewer provided freely for CIGI testing (see [here](#)).

It is provided in `/Runtime/CIGI/MPV`

Go to the `/Runtime/CIGI/MPV` directory.

Edit the following file: `config/system.def` and change the `host IP address` accordingly (should be the IP of the machine which is running MPV)

Then, start it using `start_MPV.bat` batch file

Now, open vsTASKER and load the `test_MPV` database in `/Db/CIGI/test_MPV`

Go to **CIGI** and double click the background (or right click, then **Properties**). Change the IP with the one running MPV.

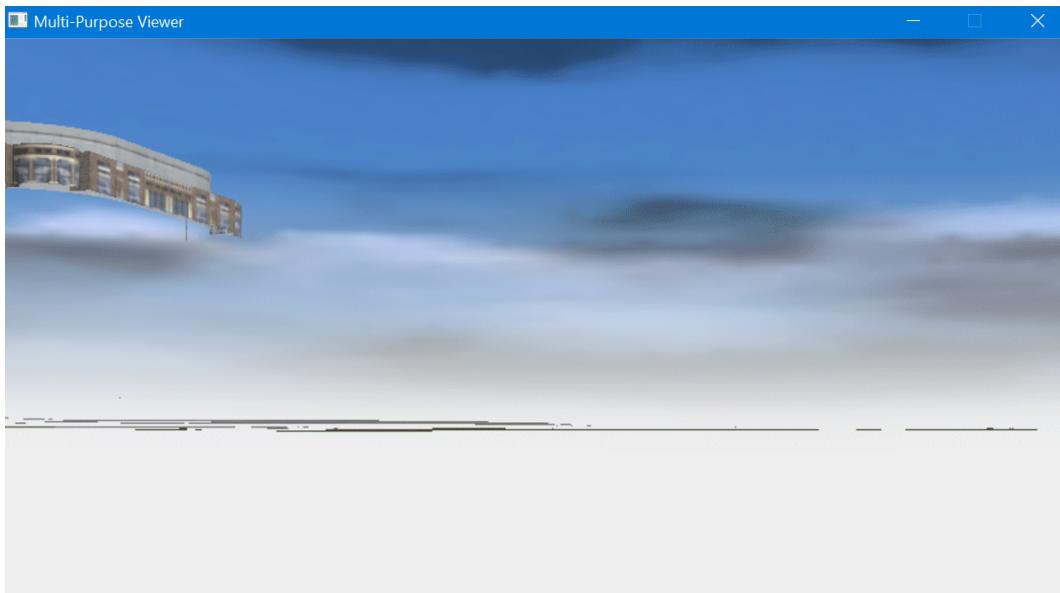
Recompile and load the simulation.

You should see the following on the console:

```
Having loaded environnement for: test_MPV
Initialize Shared-Mem...
Setting up Networks...
Initialize Networks...
Initializing ports to CIGI:
192.168.0.22: Host -> 8004 -> IG -> 8005 -> Host
Successfully connected to CIGI IG server
Force reset IG (Asynchronous) with database 1
Waiting for IG to Operate...ok
Initialize Logger...
Initialize HMI manager...
Initialize SQL...
Initialize Global...
RTC initialized successfully
Simulation Engine is ready
```

and the MPV should show this image:

## Testing with MPV

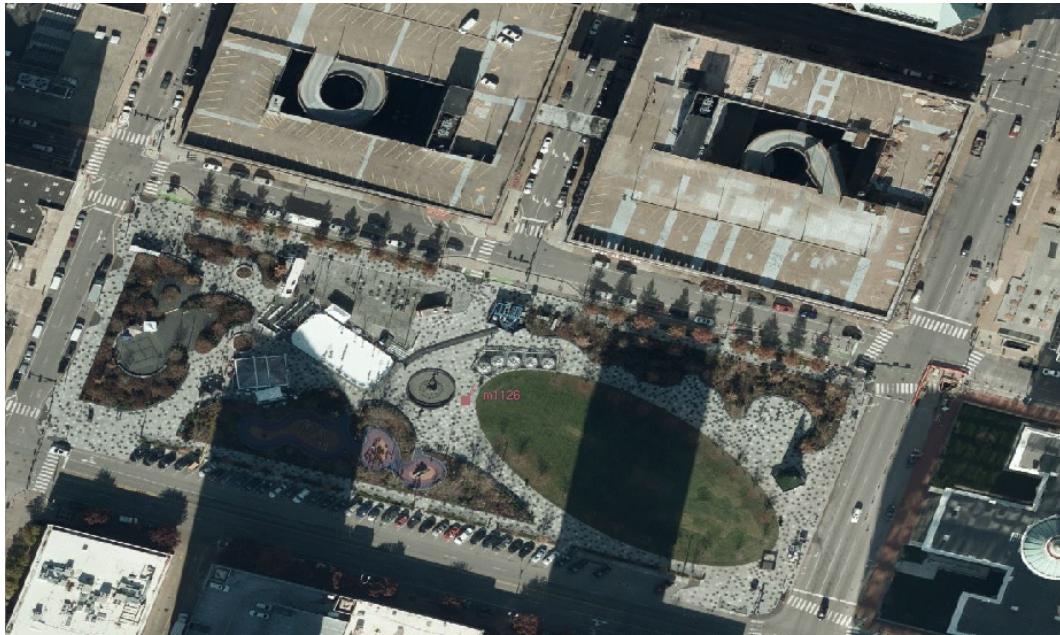


Now, start the simulation and select the **F15** on the map. The MPV should focus on the aircraft:



Then zoom on the map, click anywhere (to discard the focus), and select the red **m1126**:

## Testing with MPV



The MPV should now show the land vehicle on the park :



## **Testing with MPV**

# Using VBS-IG

With this step by step procedure, you will learn how to connect vsTASKER with VBS-IG in very few time.



*It is advisable to use a gamepad (or joystick) to control the camera in VBS-IG, as the mouse cannot be used.*

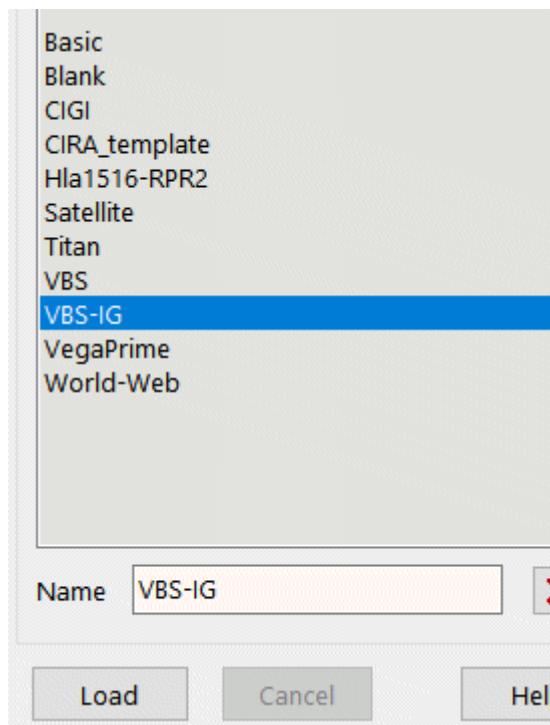
- **vsTASKER Side**

The first thing to do is to create a vsTASKER database using the VBS-IG template.

Do the following:

**File::New**

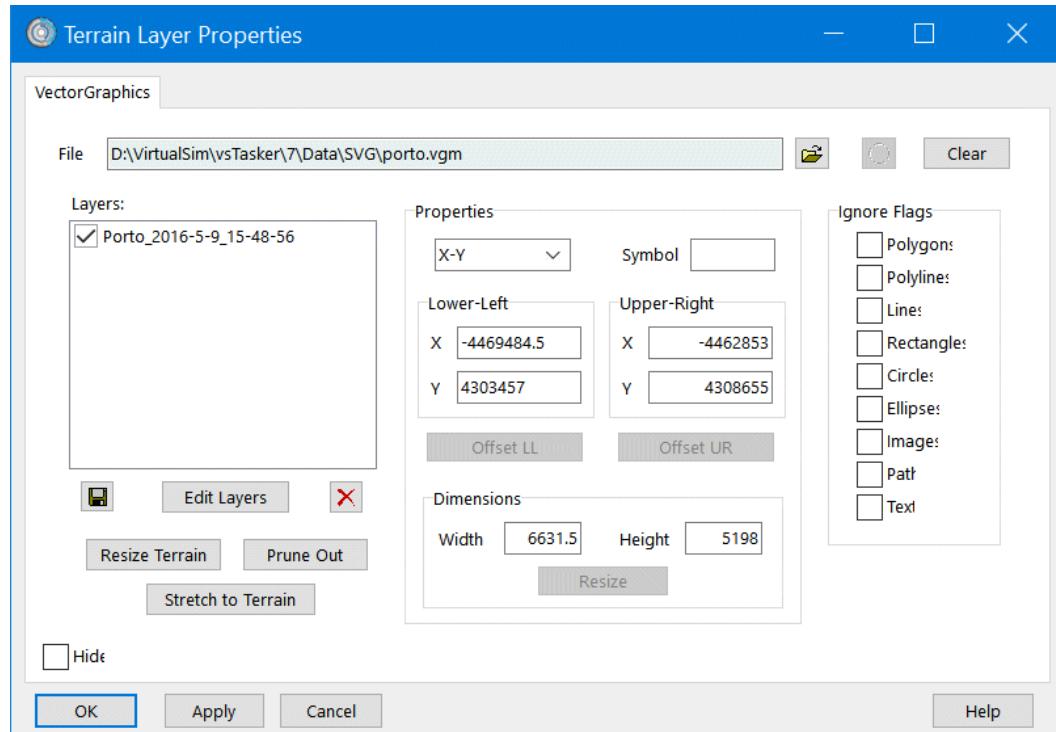
Then select CIGI template:



Save then the database under the name you want (ie: [test\\_vbs\\_ig](#))

## Using VBS-IG

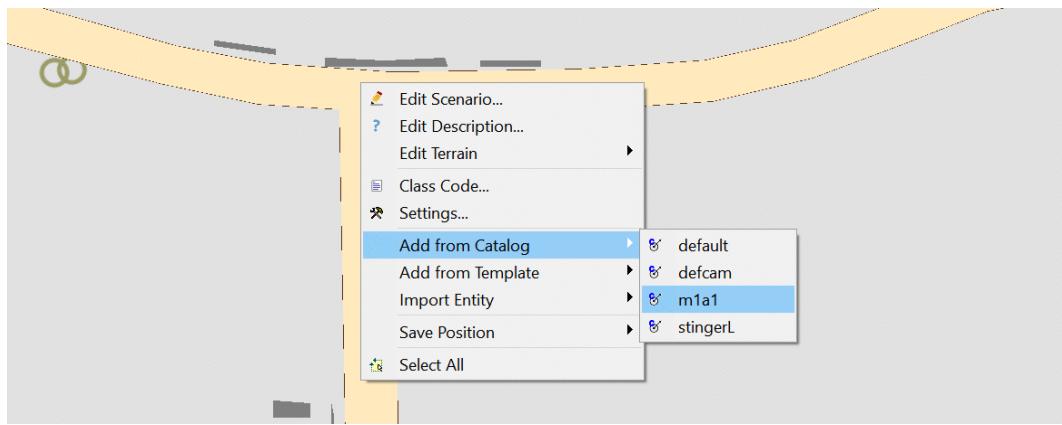
Load the **porto** database. In **Environment::Terrain**, select **VectorGraphics** icon (greyed out) and load **porto.vgm**



**Resize Terrain** then **OK**



Now, on the map, put a M1A1 tank on the road. Zoom, right click and **Add from Catalog** the **m1a1** entity.



Compile and run.

Once the IG is ready, select the tank on the vsTASKER map and see the focus on the IG



If you have a Gamepad connected, use the thumb stick to move the camera around and the left and right triggers to zoom in and out.

## • Troubleshooting

If there is no connection, make sure that the ports are correctly setup.

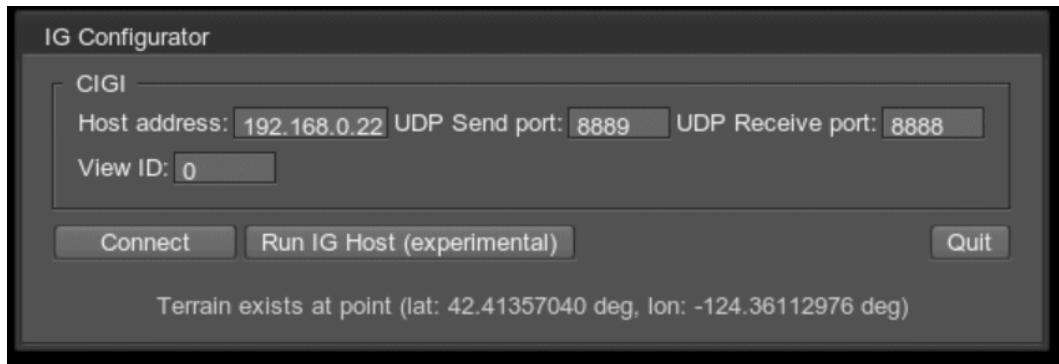
## Using Unigine

# Using Unigine

Install *Unigine 2 Sim Kit*.

Install [CIGI Demo](#) (using the *Unigine SDK Browser*)

Run the demo.



Set the IP to the [local machine](#) (127.0.0.1 may not work, use the real IP)

Set **Host receiving** port to [8889](#) and the **IG receiving** port to [8888](#).

Then Connect

Open vsTASKER.

Load the [demo\\_cigi](#) database in [/Data/Db/CIGI](#)

Compile the database (or just load the SIM if already compiled)

vsTASKER console should display the following lines:

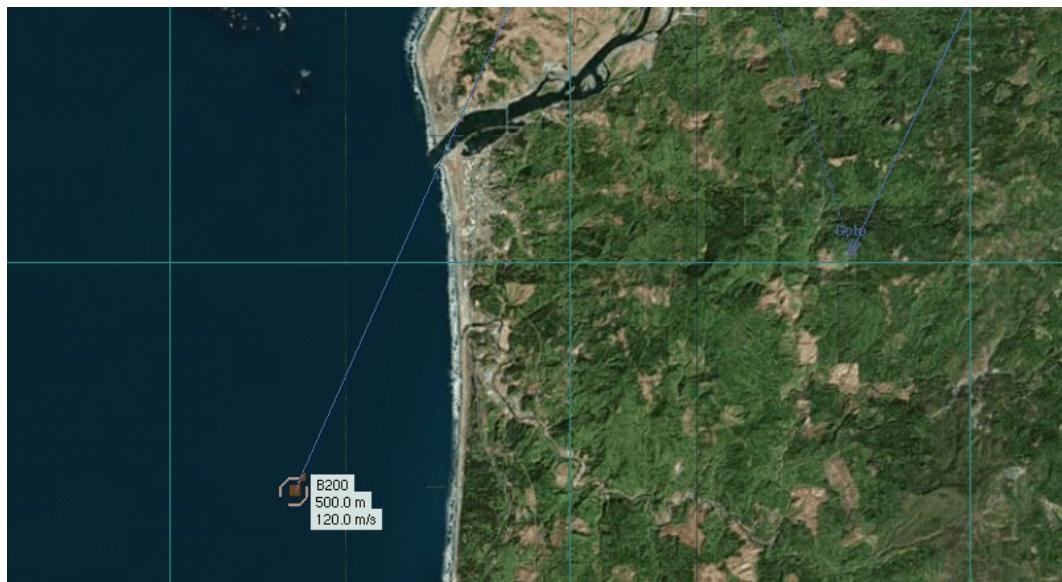
```
Having loaded environnement for: demo_cigi
Initialize Shared-Mem...
Setting up Networks...
Initialize Networks...
Initializing ports to CIGI:
192.168.0.22: Host -> 8888 -> IG -> 8889 -> Host
Successfully connected to CIGI IG server
Waiting for IG Ready...
Operate IG (Asynchronous) with database 1
IG_Control ig_mode(1) db(1) ig_frame(1175) host_frame(1)
Initialize Logger...
Initialize HMI manager...
Initialize SQL...
Initialize Global...
RTC initialized successfully
Simulation Engine is ready
IG_Control ig_mode(1) db(0) ig_frame(1179) host_frame(12)
```

Unigine Console should display the following lines:

```
---- CIGI info ----  
Host address: 192.168.0.22  
Send port: 8889  
Recv port: 8888  
Version: 33  
Packet size: 1432  
  
Script loading "cigi/cigi.cpp" 31ms  
World loading "cigi/cigi.world" 126ms
```

Then start the simulation.

When the B-200 aircraft is flying and simulation RTC is running at 30hz, click on the aircraft symbol.



The IG now has the camera focused on the aircraft. If you are using a **gamepad**, just use the triggers to zoom in/out.

Left thumb stick moves the camera around.

## Using Unigine



# LAN

- **Concept**

- **How to Use**

Each message sent or received through a **SockItem** must have a predefined header made of **two integers** (so **8 bytes**), organized as this:

```
unsigned short source, type;
unsigned int size;
```

**Source** is the any value used to qualify the message provider (is several devices are writing into the same port).

**Type**: specify the type of the message for casting it properly

**Size**: is the total length in bytes of the message, for reading the buffer correctly.

So, all user data structure that must be passed or read through a SockItem must be defined as a class inheriting from a base class (**MsgData**) of vsTASKER.

This base class can be found in [/include/engine/vt\\_sockets.h](#)

For example, if a message must be made of a character string and a value, it should be defined this way:

```
class myMessage : public MsgData
{
public: myMessage() { size = sizeof(*this); type = 123; }

char label[20];
float value;
};
```

Then, sending such a message would be done, from a SockItem code, like that:

```
MyMessage my_mess;
strcpy(my_mess.label, "foo");
my_mess.value = 12.332;
send(my_mess);
```

And receiving such a message from outside would be done, in a SockItem, like that:

## LAN

```
if (msgData->type == 123) {  
    MyMessage* my_mess = (MyMessage*) msgData;  
    printf("%s = %f\n", my_mess->label, my_mess->value);  
}
```

## **Integration with third-parties**

vsTASKER can easily be integrated with numerous other software used in the simulation industry. Instead of reinventing the wheel, vsTASKER provides gateway or built-in integration with some tools widely used by the community.

Here are some of them.

# **VBS-IG**

This tutorial has been tested with VBS-IG 12.2

You need to have the product licensed from Bohemia Simulation.

The purpose of this tutorial is to learn how to create a simple scenario on vsTASKER and display it on VBS-IG.

In this exercise, a tank will detect a moving stinger vehicle and will destroy it with its main gun.



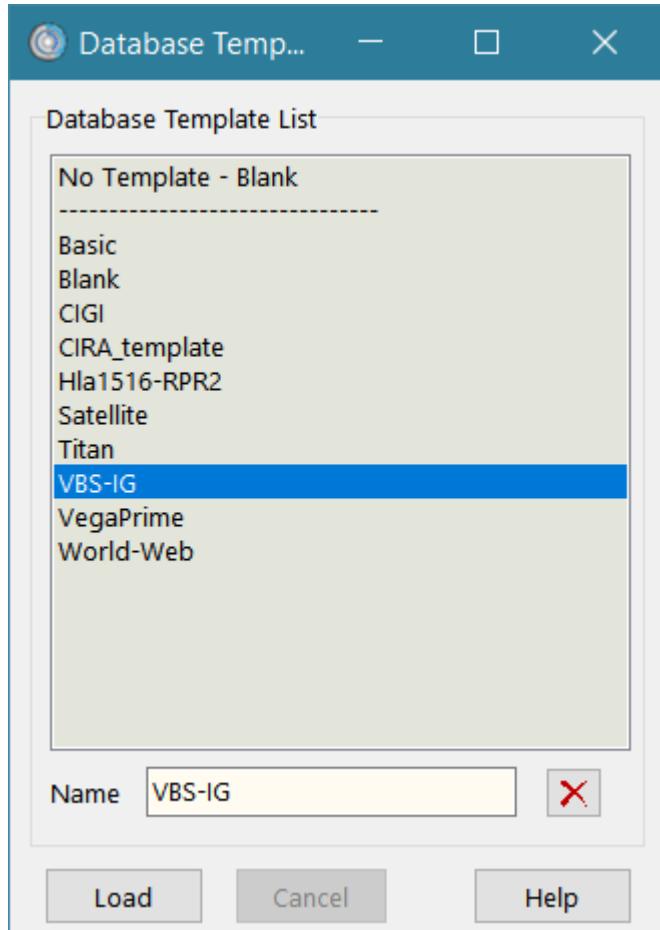
*Specific license is mandatory. Contact VirtualSim if you need the CIGI module.*

# Scenario Setup

The simplest way to create a VBS-IG scenario from scratch is to use the VBS-IG template.

It will load the necessary environment, including the CIGI packets and the CIGI model container.

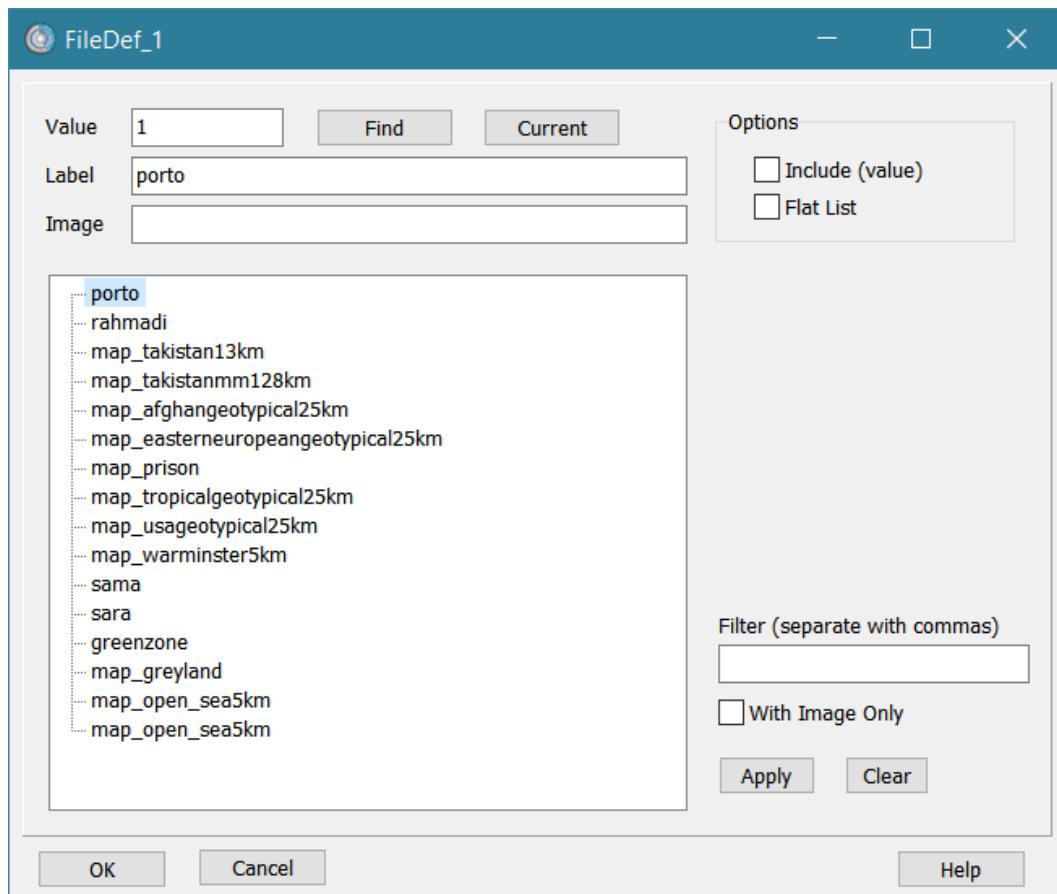
In vsTASKER, **File::New** to create a new database:



Select **VBS-IG** then **Load**

Now, select a terrain. Let's take Porto from the list:

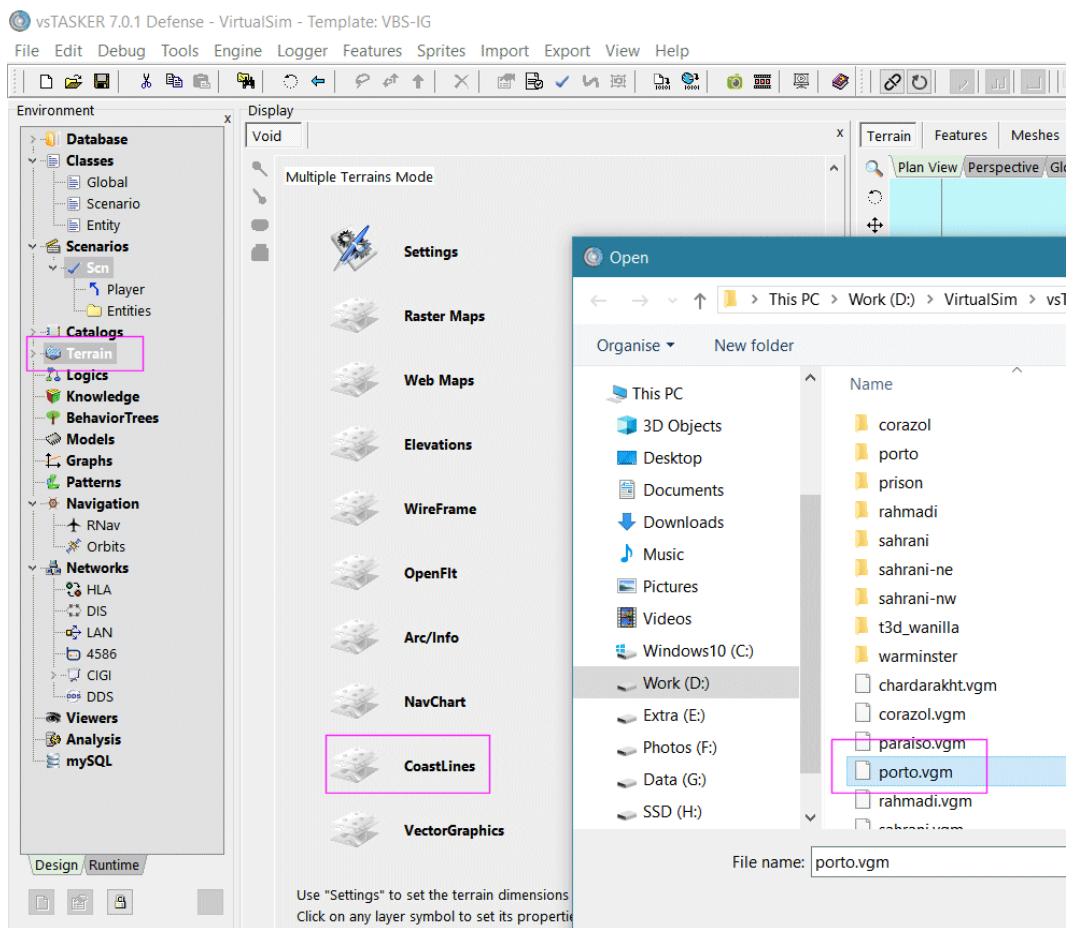
## Scenario Setup



Time to load the corresponding terrain.

Click on [Environment::Terrains](#) then [VectorGraphics](#) and select [Porto.vgm](#) :

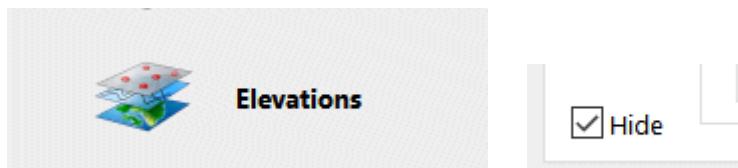
## Scenario Setup



Elevation terrain is also important as the SVG do not include them and requesting Height of Terrain by CIGI for each entity can become too network and CPU intensive. As most of the time, height of a terrain do not change, this could be done before going runtime.

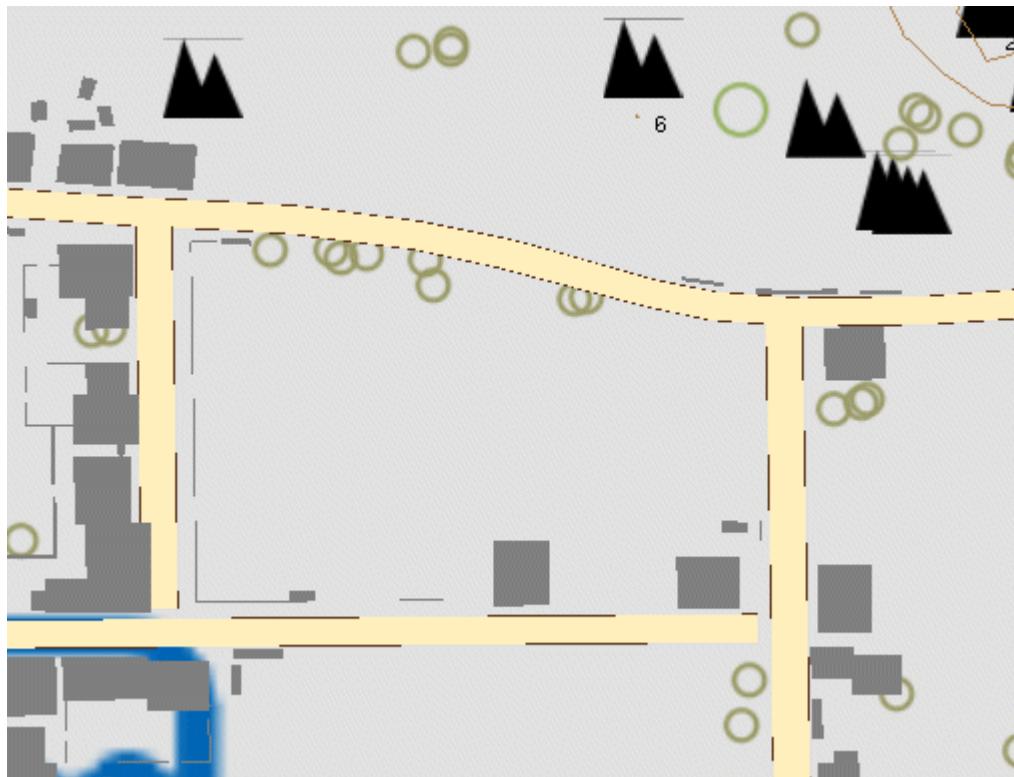
Select **Elevations** layer and load **porto.elev**.

You can keep the elevations visible or **hide** the layer to only keep visible the SVG map.

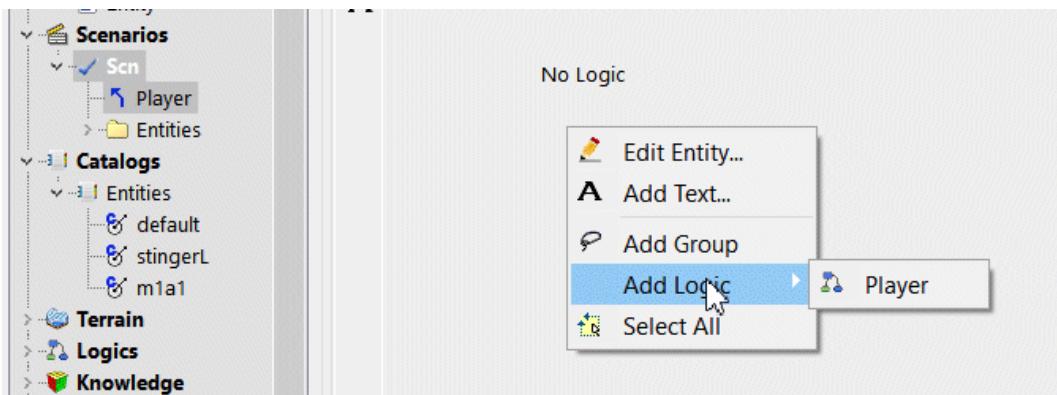


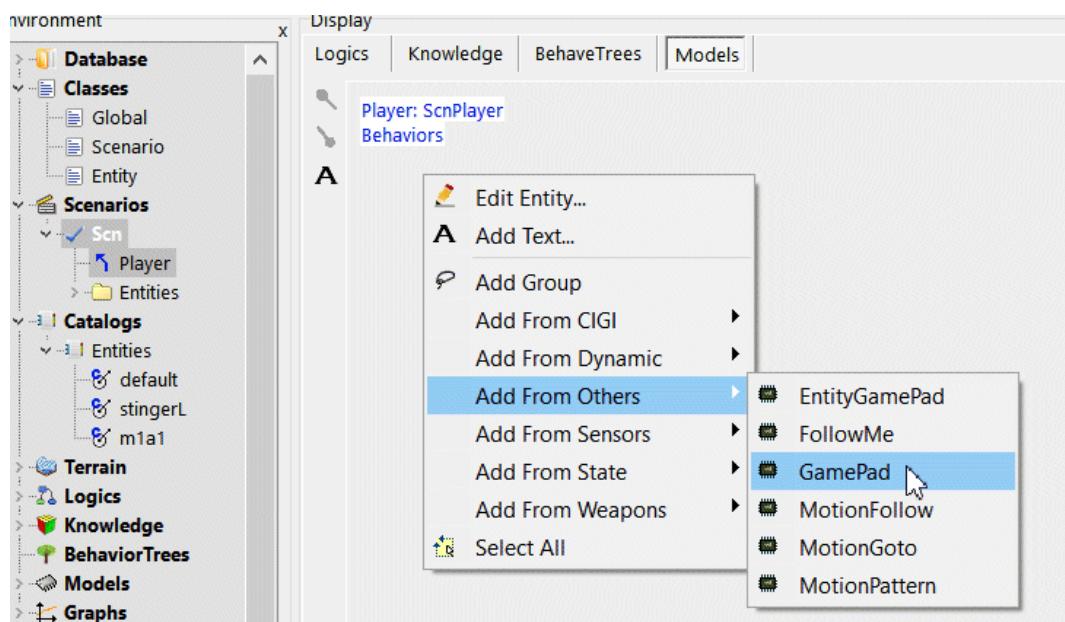
Now zoom in this area and save the database under whatever name you want.

## Scenario Setup



Now select the Scenario Player and give it the **Player** logic and the **Gamepad** component:





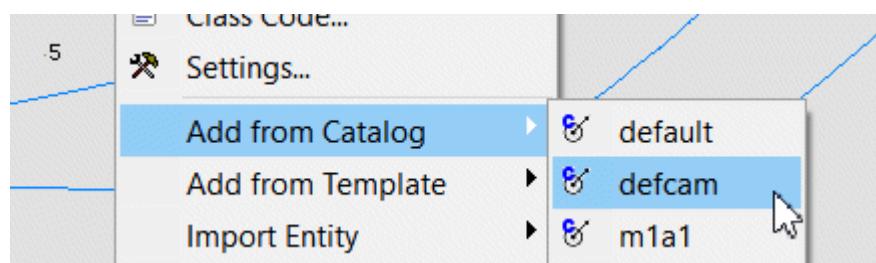
## • Default Camera View

It is mandatory to give the camera a default focus when nothing (no entity) is selected.

This can be any entity of the scenario or a particular one.

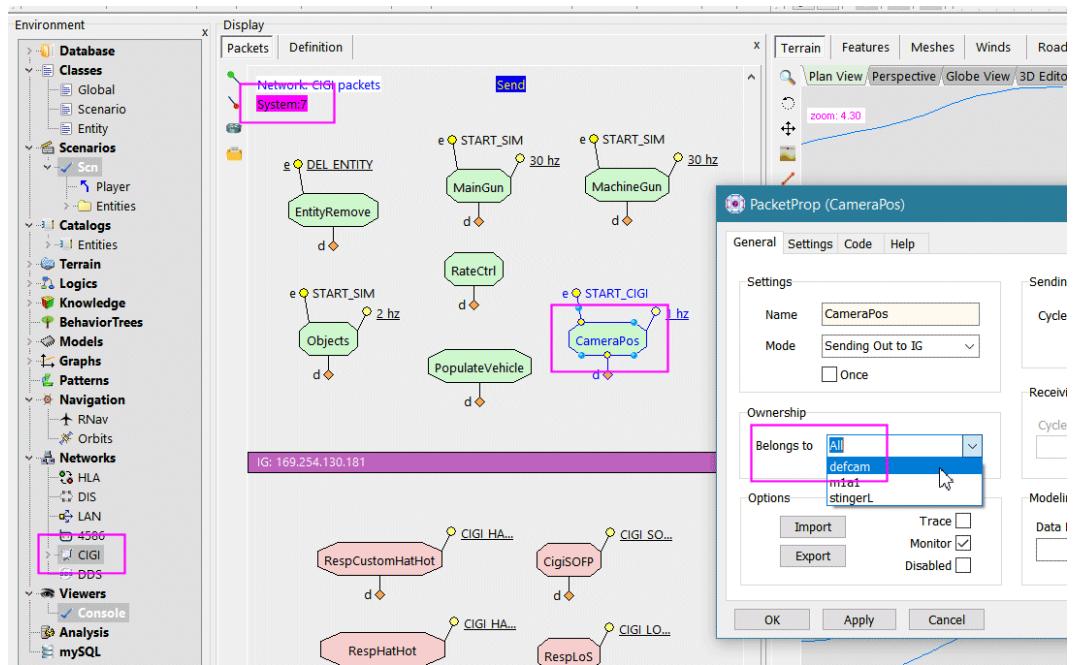
In the Catalog, there is a **defcam** (ghost) entity which serves this purpose. You can instantiate it anywhere on the scenario. The IG camera will look at this point at startup.

Create this entity on the map:



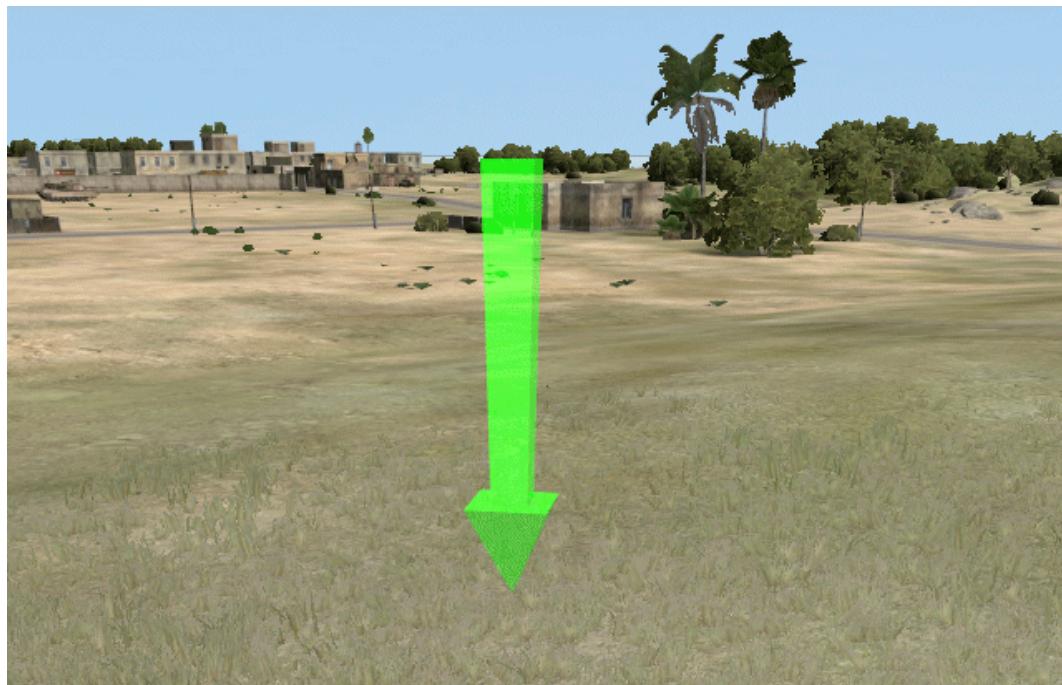
Then, open the **CameraPos** Packet which is in the System folder of the **Environment::Networks::CIGI** category:

## Scenario Setup



If you have added the `defcam` entity, select it from the list. You also could have selected `m1a1` so that to have the camera directly focusing the tank at startup without the need to select it on the map.

Below, camera focused on `defcam` entity. You can drag it on the map to explore the area in 3D (this works best with 2 screens).



# Map Extraction

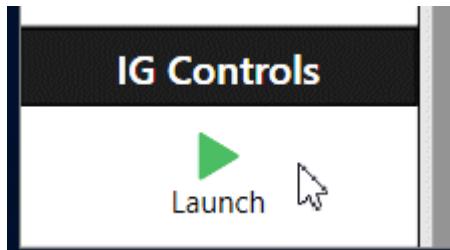
In this chapter, we are going to see how to extract a SVG map from VBS-IG and import it into vsTASKER.

VBS-IG (like VBS) can only export a map (full or partial) as an SVG or PDF format. The PDF is useless for us.

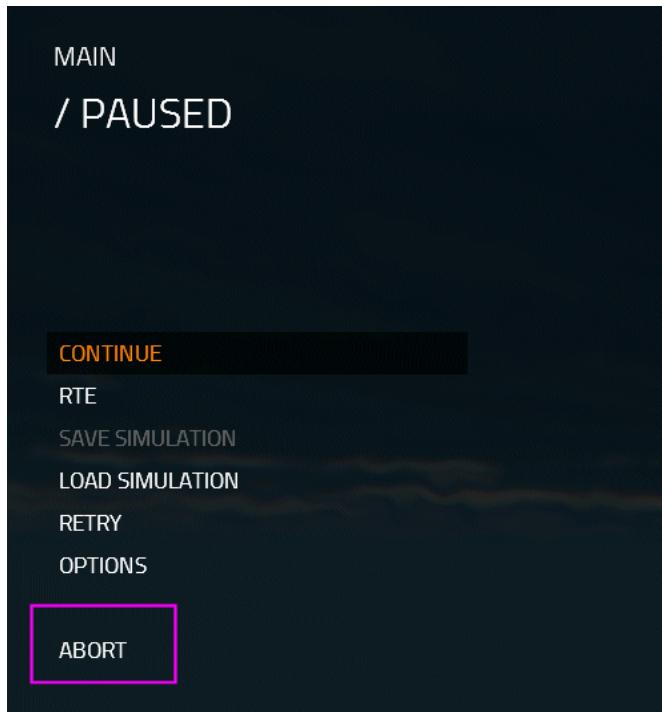
The SVG can be used as vsTASKER provides an SVG importer and converter (see the **User Manual** for more information).

## • **SVG Export from VBS**

Start VBS-IG from the Studio:



In the IG, press **ESC**, then, in the following menu, select **ABORT**:

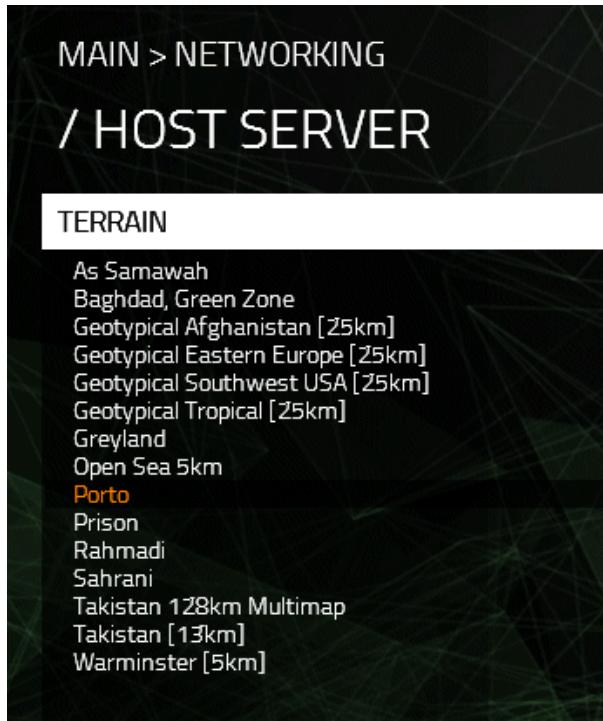


## Map Extraction

Then from the Debriefing page, select **CONTINUE**:



And once in the Terrain selected, chose [Porto](#) (for our example), then **START**:



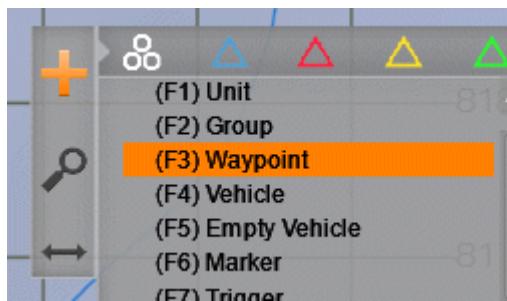
Now, you are in the VBS editor.

Unfortunately, VBS programmers never thought someone would be interested in getting the coordinates below the cursor. Coordinates are displayed in a useless format, but forget about having the Lat/Lon/Alt coordinates copied into the clipboard, from a right-click menu. This exists only on vsTASKER so far.

So, we need to process the ugly way: by creating two objects to materialize the bottom-left and top-right corners of our area of interest. This is mandatory to get later the Lat/Lon coordinates as they will be requested by the map extractor.

Let's select Waypoint:

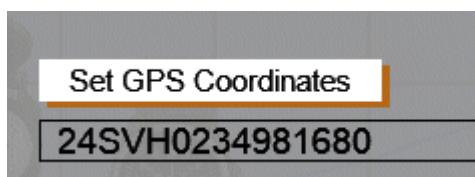
## Map Extraction



and add the two waypoints at the corners of the area we want to extract:

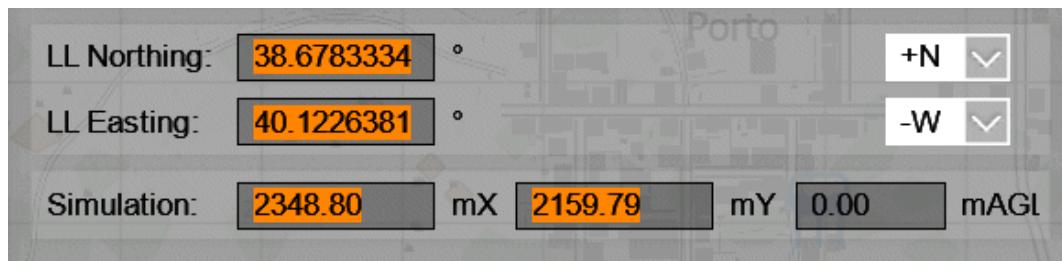


Now double-click each waypoint and select **Set GPS Coordinates** from the *Object Property Window*



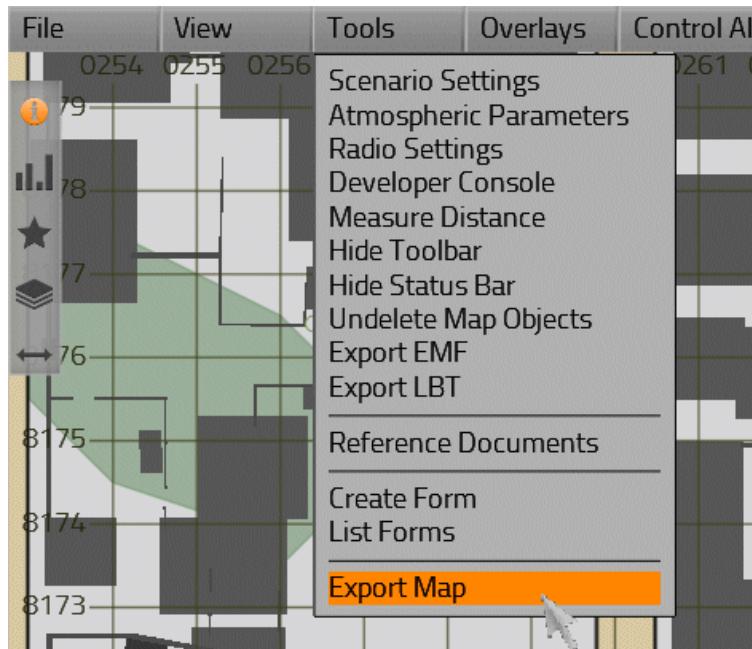
From the next window, write down the two decimal Lat/Lon (do not forget the sign, - W = -40) and the simulation coordinates (always useful although we don't need it for now).

## Map Extraction



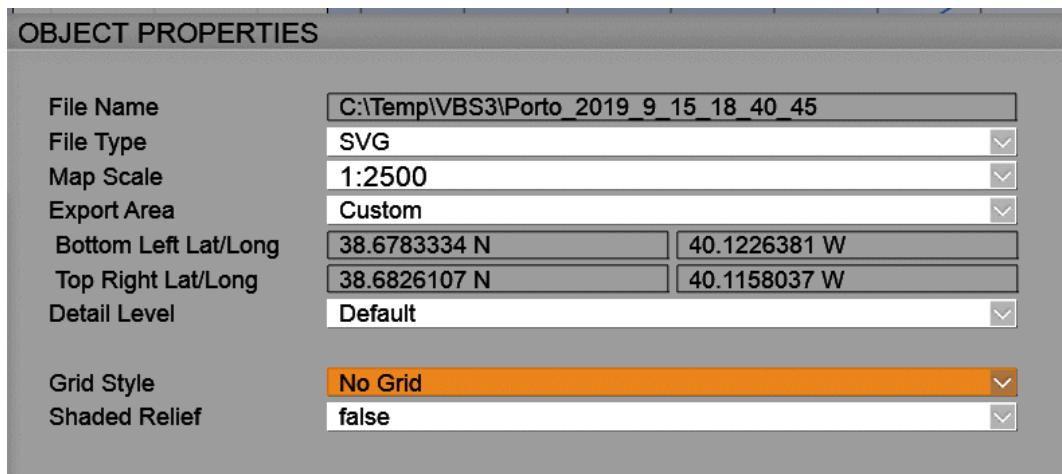
Do that for the two corners.

Now, in the menu, select **Tools::Export Map**:



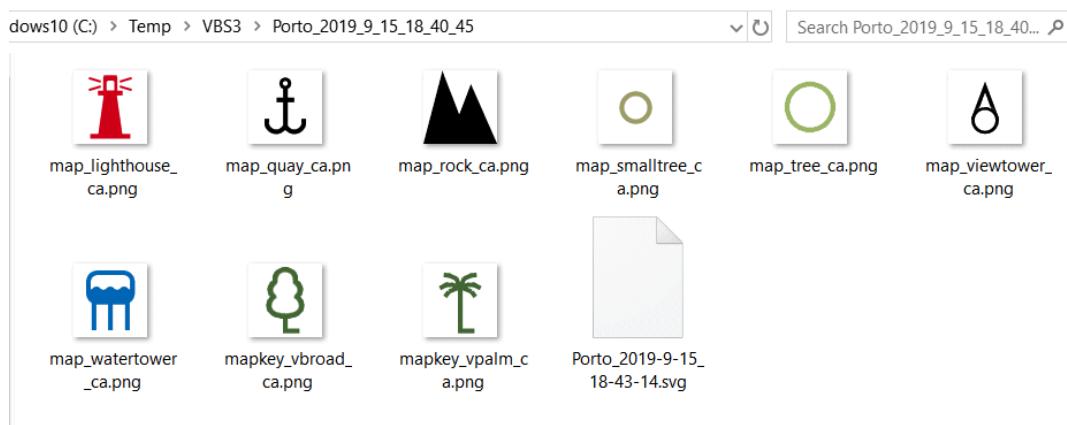
Then, in the extractor window, setup the fields like below (do not forget to paste the coordinates of your corners). The higher the scale the better. Detail level is up to you.

## Map Extraction



Then press **OK**.

Now, you have your **svg** file in **C:\Temp\VBS3**:



If you open it (in a text editor) you will notice that the coordinates in Lat Lon are lost. You have only the simulation coordinates mentioned:

```
<svg width="24.011cm" height="18.6957cm" viewBox="2348.8 2159.8 600.276  
467.392"
```

## • Importing the SVG into vsTASKER

Now that the SVG raw file is exported, we will import it into vsTASKER and, most importantly, put the map at the correct position so that there will be no discrepancies during the simulation.

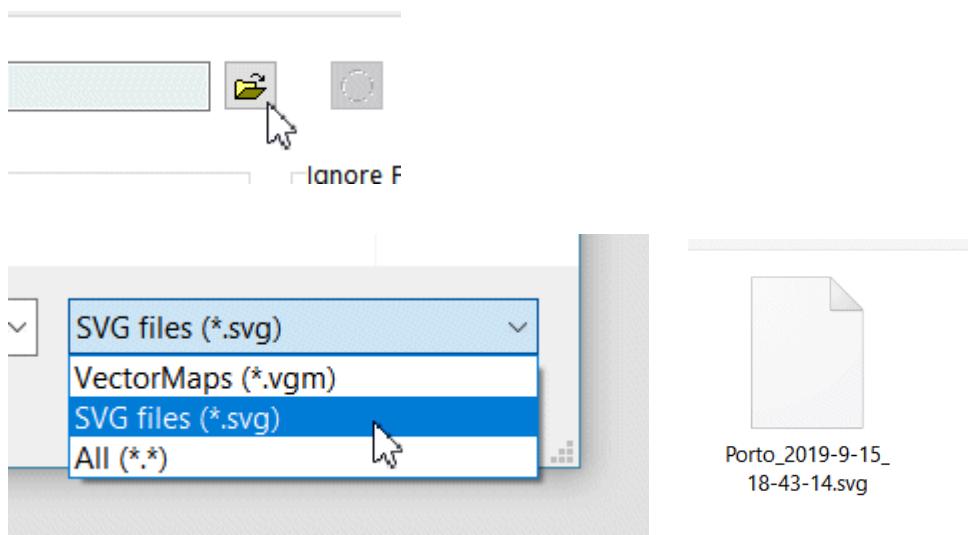
As we are going to use CIGI, simulation coordinates will be of no use. We need to position the map at the correct Lat Lon coordinates.

## Map Extraction

In Environment::Terrains, select VectorGraphics Layer:

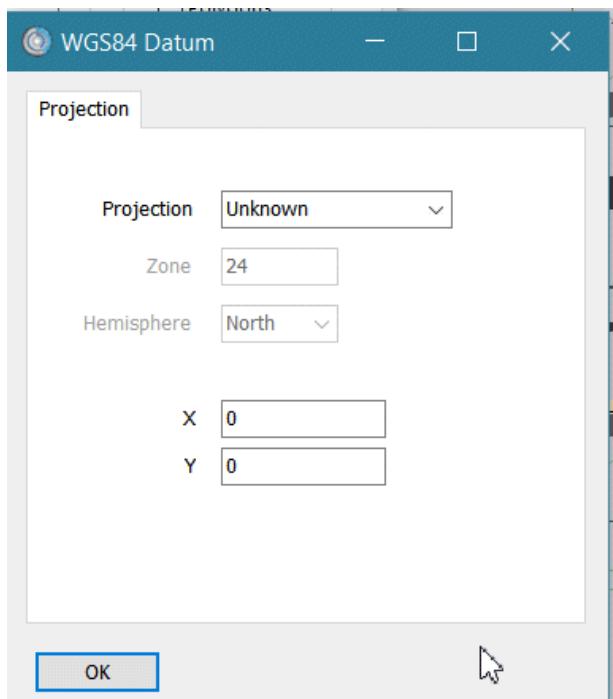


Then load the exported SVG file from C:\Temp: (do not forget to change the filter type from vgm to svg)

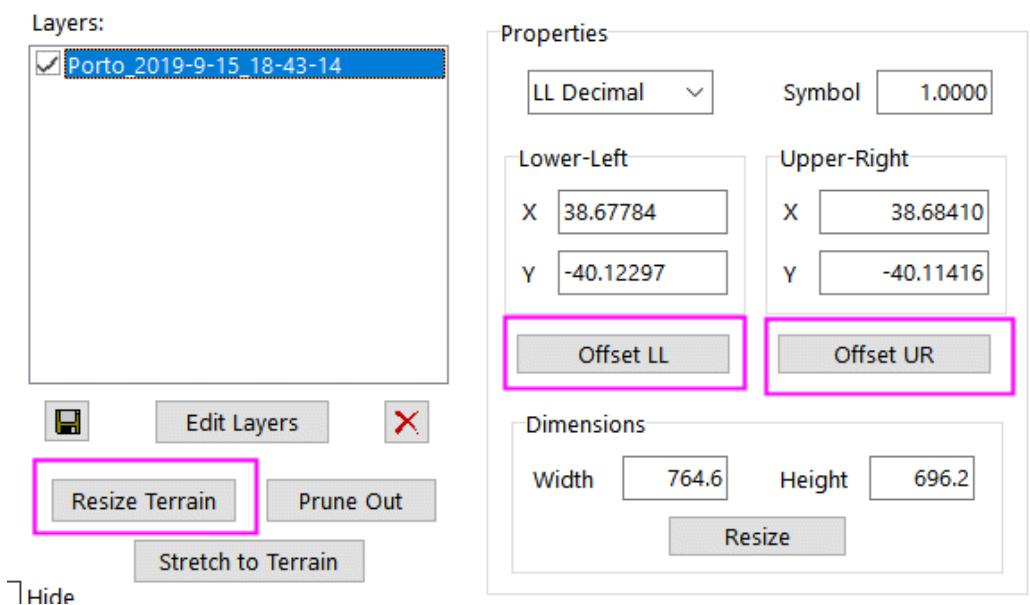


Then click

and leave the projection as Unknown:



Once imported, select the layer, drop the corner coordinates (lower-left and upper right), then **Offset LL** and **Offset UR** corners and do not forget to **Resize Terrain** so that to see something.

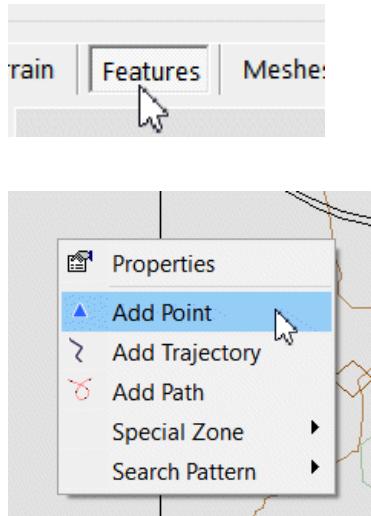


In an ideal world, this would be enough.

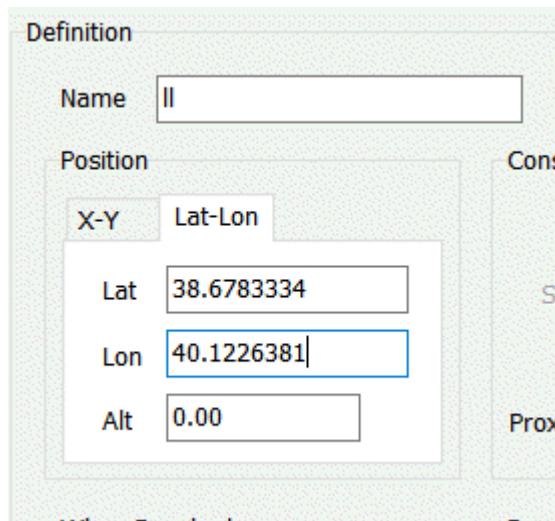
## Map Extraction

But, as VBS is not exporting a map which matches the specified corners (and waiting for a fix is not going to help), we need to fine tune the alignment with a turn around.

For that, we will go to **Feature** mode and create two Feature **Points** at the exact corners coordinates:



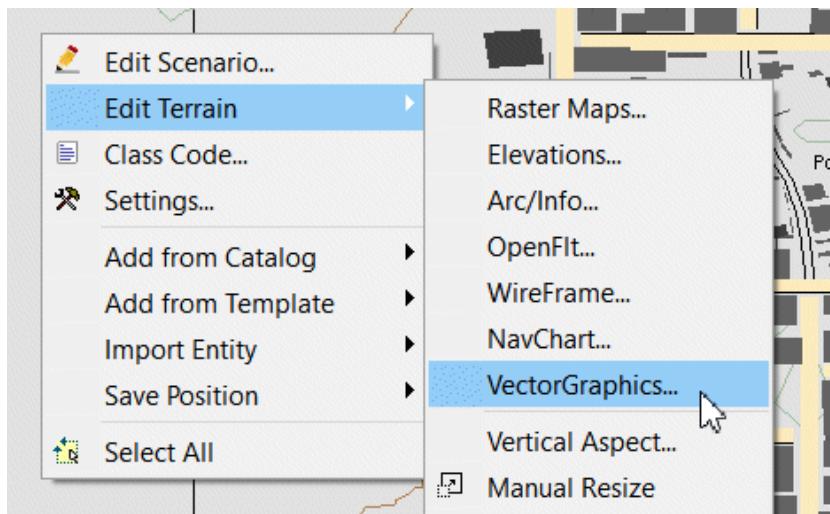
Create a point anywhere, name it LL (for Lower Left corner), double-click it and enter the Lat Lon coordinates you gave to VBS for this corner:



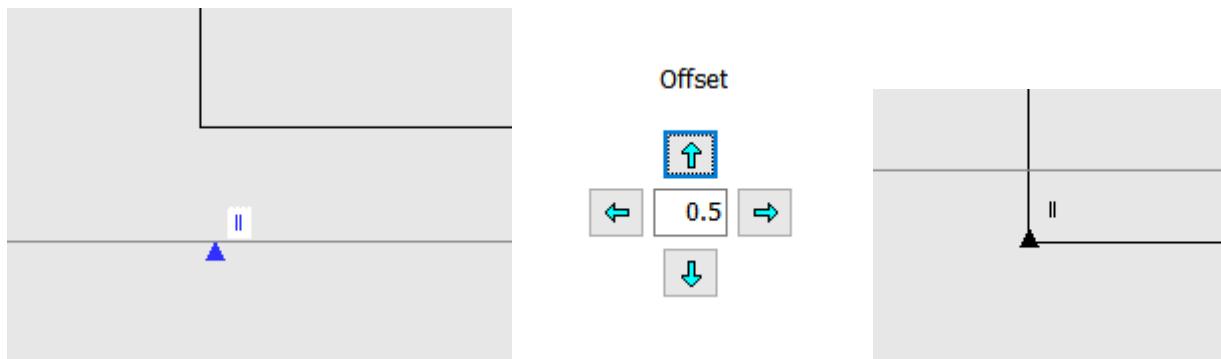
Once you have both points located on the map, the SVG layer must be offset (and maybe rescaled) to match these positions.

The exported map has normally a black contour line which ends at both corners. We are going to use that guide for this operation.

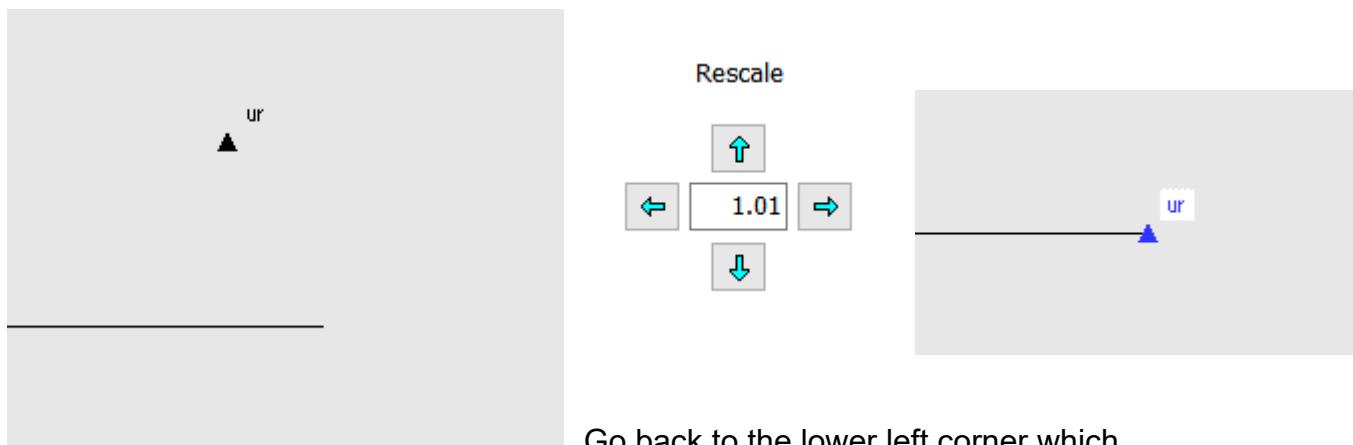
Go back to **Terrain** mode and bring the rescaling window for SVG layer:



Now zoom at the Lower Corner point and use the offset arrow to put the line on the point (you can change the increment value):



Now go to the Upper Right corner, zoom again, and use the rescaling to put the terrain corner on the point:

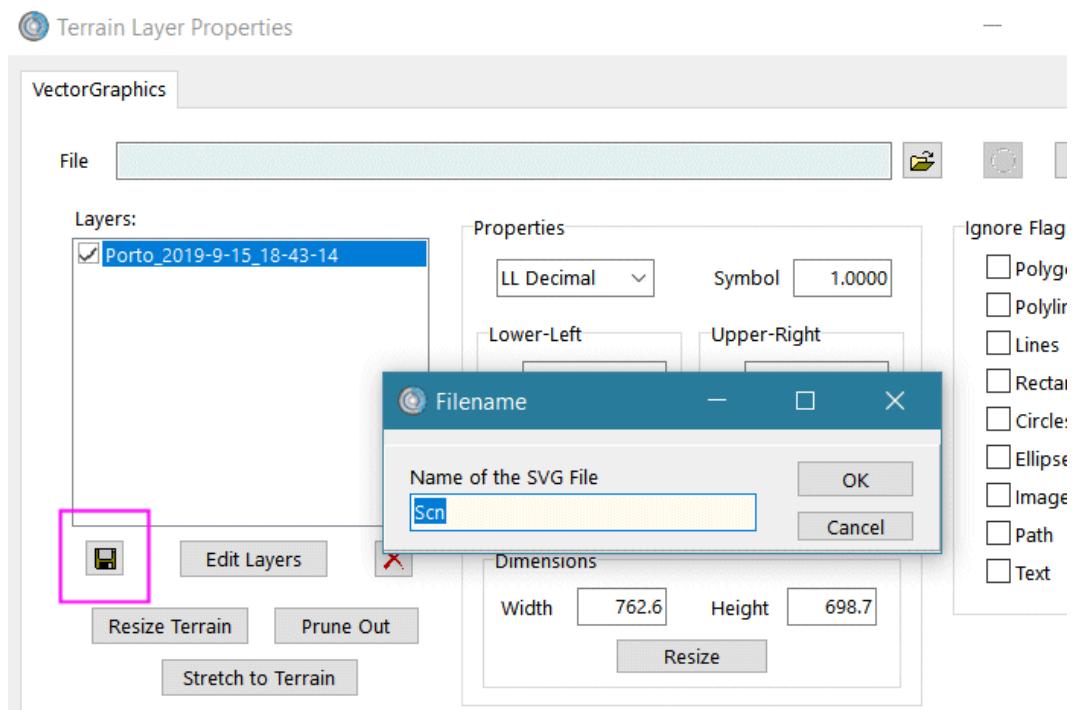


Go back to the lower left corner which may have shift a bit during the rescaling, repeat the process until both are aligned.

## Map Extraction

You are now set.  
The SVG is correctly imported.

Now **save it !**



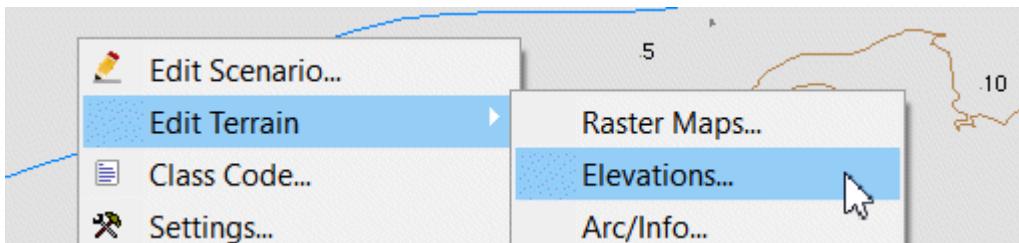
Press **OK** and the file is now stored under the given name in [/Data/Svg](#)

# Elevation Layer

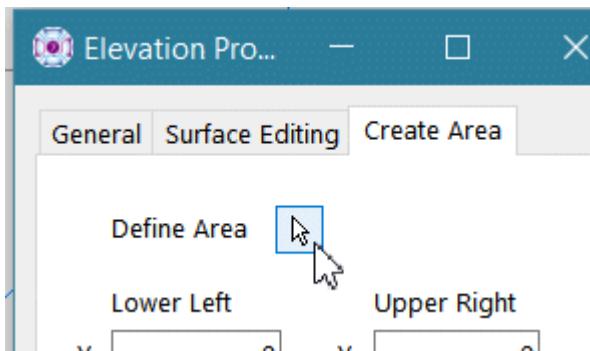
It is possible to create an elevation file made of several layers and use a terrain processor (online queries, OSG or CIGI engine) to provide the height of each plot. In our example, even if the `porto.elev` elevation file is available, we are going to build two layers at different locations.

Layers can be separated or on top of each others. It is advisable to start with the widest low level one and finish with the higher definition one on top of the list.

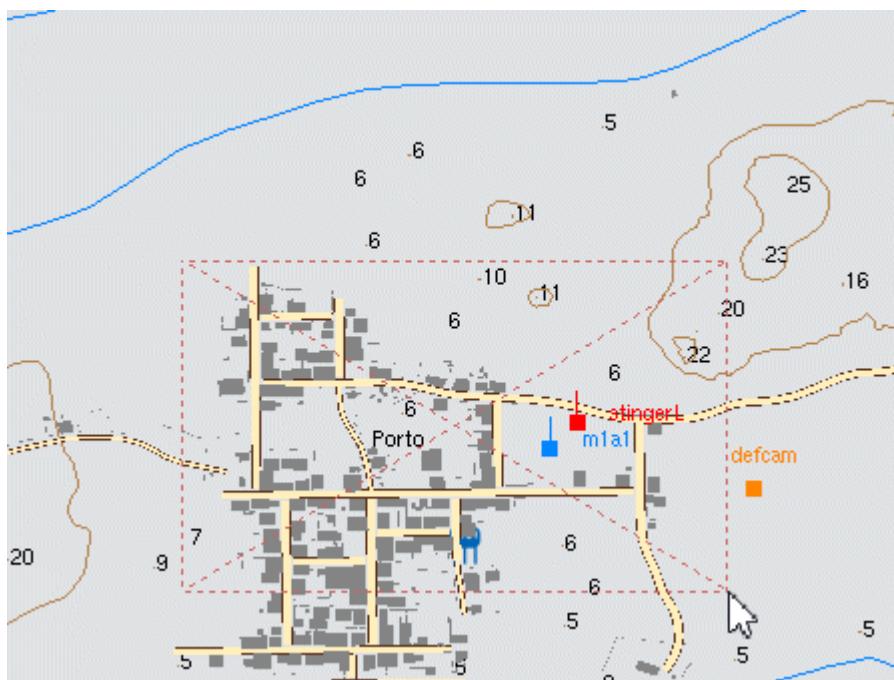
First, open the **Elevation Editor** (right click on the map background):



Then select **Define Area** button to draw on the map the elevation layer:



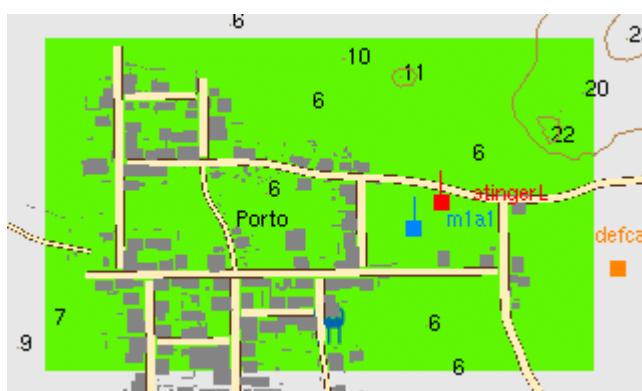
## Elevation Layer



Now, specify the **Resolution** of each plot of the layer, in meters. The lower the resolution, the higher the number of plots and the memory used:

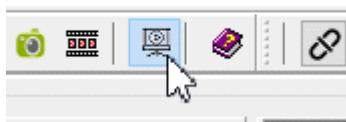
Resolution	5 m
Number of Plots	6433
Memory Size	25.1 Kb

When you are good with the resolution, press **Create**.

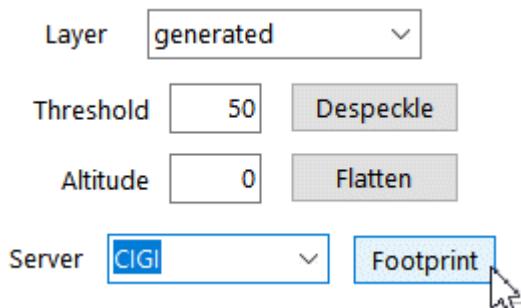


Now, we must ask the IG to provide the elevation of each plot. For that, vsTASKER is going to use CIGI messages.

Let's activate the CIGI mode for design:



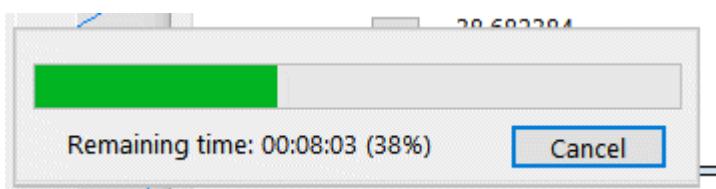
When connecting is done and entities are sent, select **Surface Editing** panel, select **generated** in the **Layer** list, then **CIGI** in the **Server** list and press **Footprint**:



*A CIGI engine (like VBS-IG but not only) must be running and be ready to receive HoT request messages and respond to host.*

The process may take some times but the good news is that this work will be saved when done.

The higher the number of plots, the longer the process will take. For big databases, you can leave it the night long and get the result on the morning.



When the process is done, zoom on the terrain. You should see something like that:

## Elevation Layer

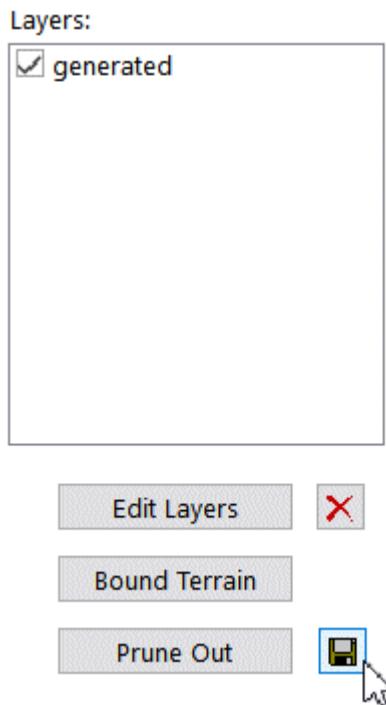


Now it's time to save the elevation database on the disk.

Select Environment::Terrain::Elevations

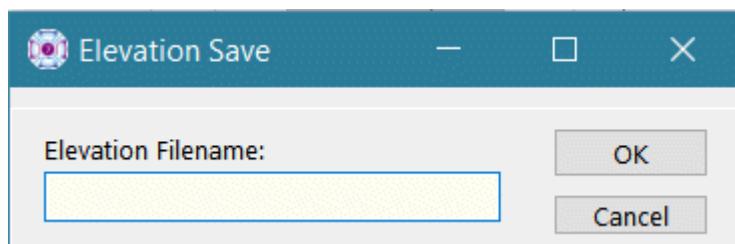


## Elevation Layer



You can see on the Elevation property window that there is no file defined but the generated layer listed.

Use the button to save it into a new elevation file.



Enter the name you want for the elevation file (which will be stored by default in **Data/Elev** directory).

And you're done.

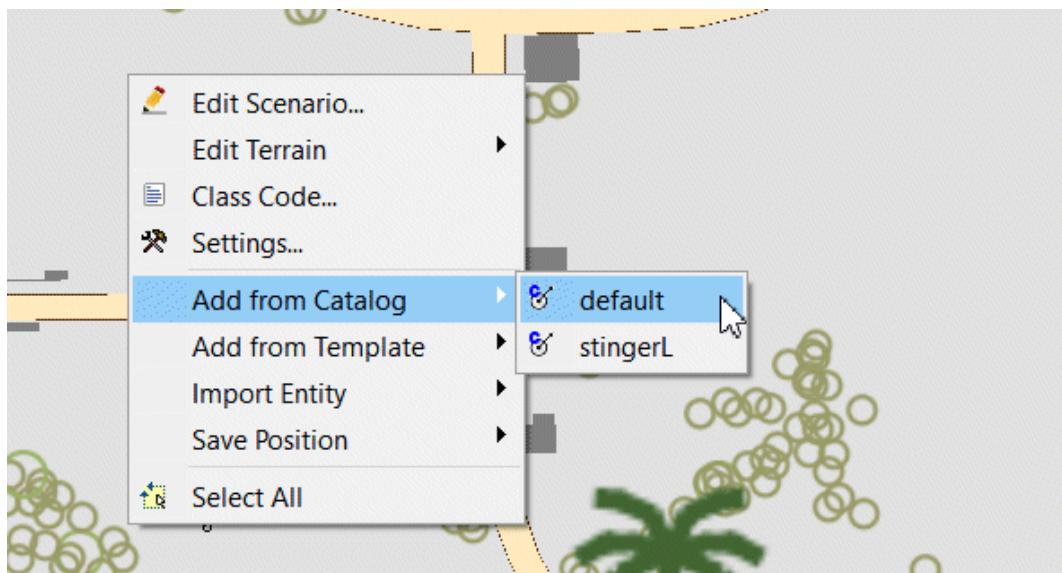
You can now save your database.

## Adding Entities

# Adding Entities

We are going to add two entities: one tank and one stinger.

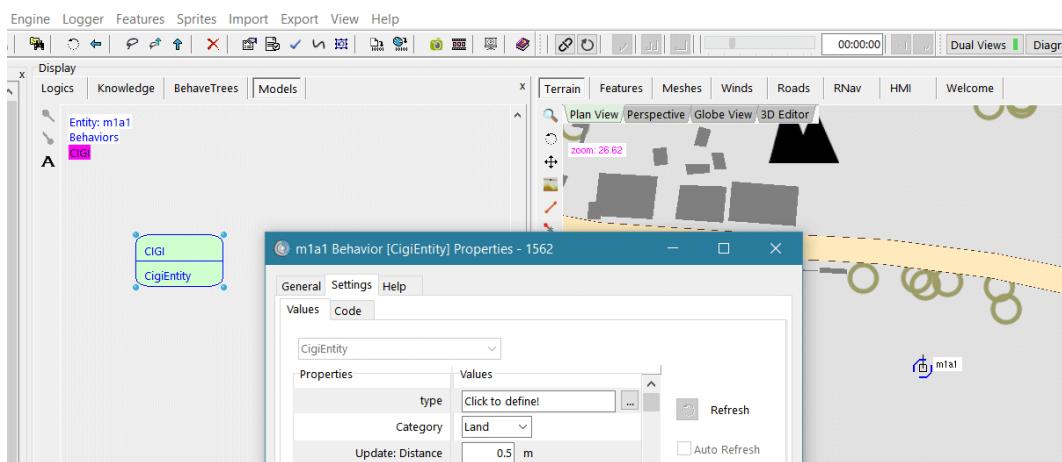
Let's add the tank in the middle of the zoomed area (previous [part](#)):



For the tank, we will use the default entity which comes with some CIGI components ready to be customized.

Once instantiated, rename it as: **m1a1**

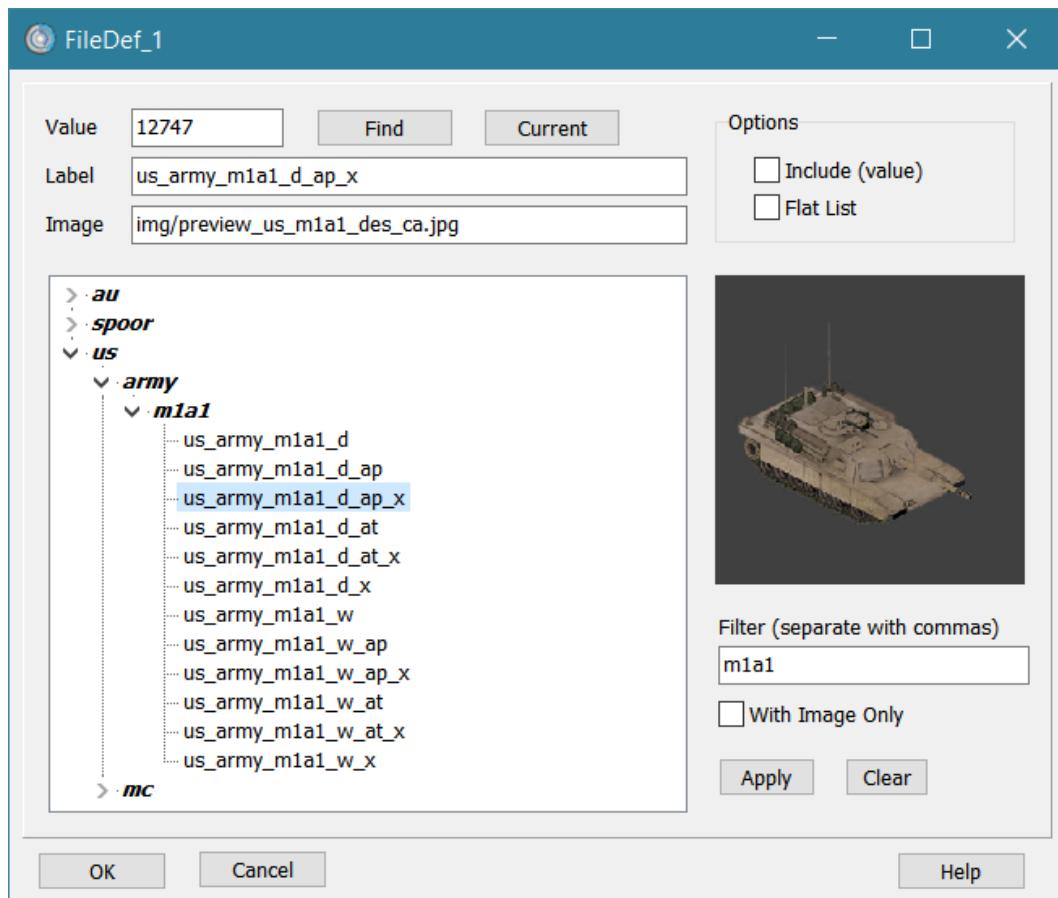
Select **Models**, open the CIGI group (double click to open) then open the **CigiEntity Data Model**:



We will define the **type** (*Click to define!*) and search in the list for the M1A1 tank. To facilitate the search in the entity database (imported from VBS-IG definition file), let's put **m1a1** in the **Filter** and click **Apply**.

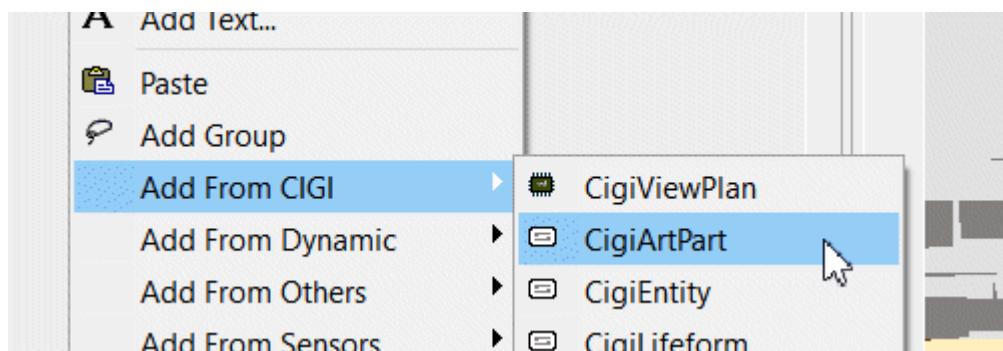
Then, open the **us::army::m1a1** and select **us\_army\_m1a1\_d\_ap\_x**:

## Adding Entities



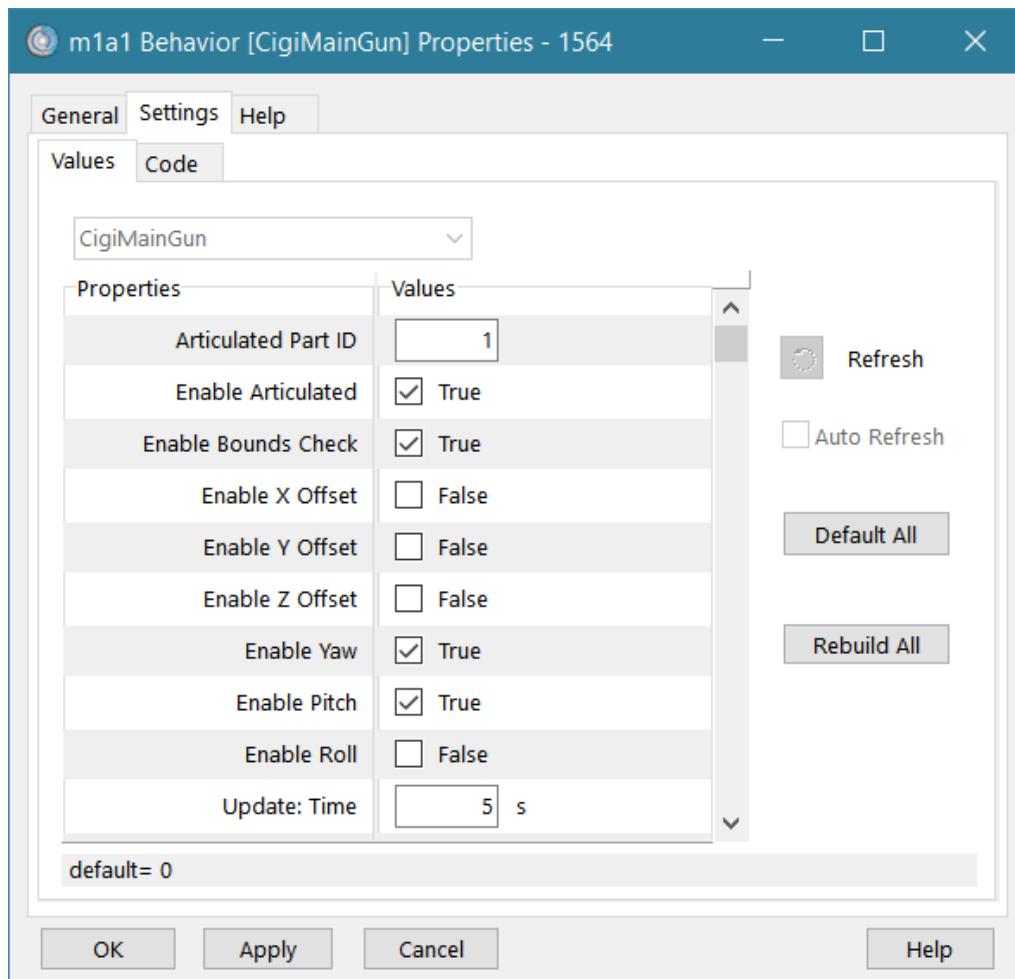
Now let's add the Articulated Part.

Right-click on the CIGI Group background and select the CigiArtPart model:

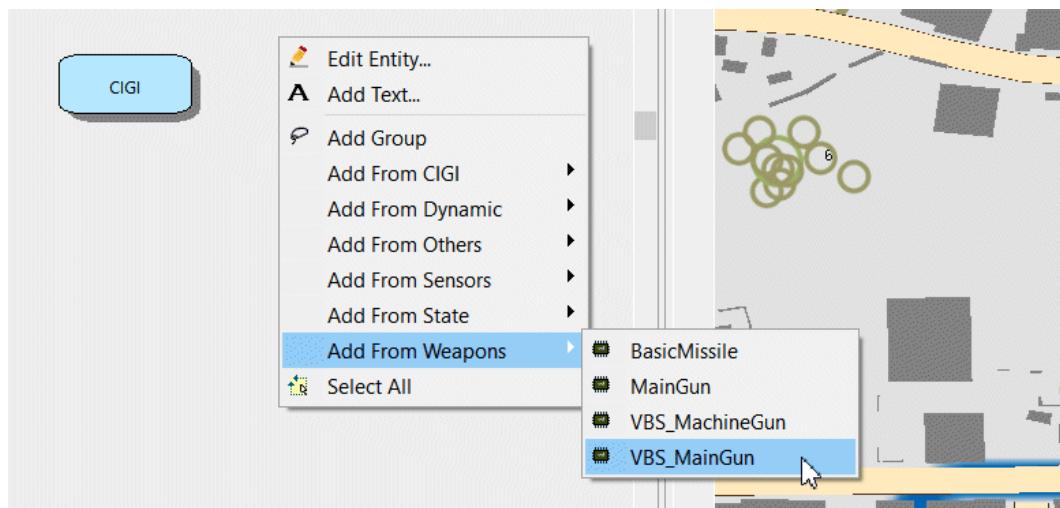


and set it this way:

## Adding Entities



Then the [VBS\\_MainGun](#) component which will use the CigiArtPart above and provides data to the CIGI packets:



## Adding Entities

Now, set it up this way:

MainGun

Properties	Values
azimuth	0 deg
elevation	0 deg
Bullet Velocity	300 m/s
show_bullet	<input checked="" type="checkbox"/> True
show_impact	<input checked="" type="checkbox"/> True
show_damages	<input checked="" type="checkbox"/> True
Show Trajectory	<input checked="" type="checkbox"/> True
drag	1
Accuracy Mode	Distance
Accuracy	10
kill_at	10 m
Damage Profile	
aware_at	50 m

default= 0

Refresh

Auto Refresh

**Default All**

**Rebuild All**

## Adding Entities

The screenshot shows the properties editor for the entity 'VBS\_MainGun'. The left pane displays the properties and their current values:

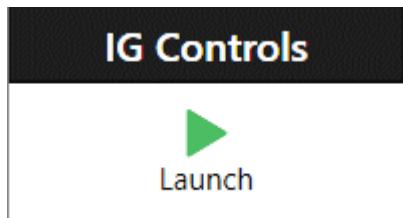
Properties	Values
Angle/sec	20 deg/s
delay	3 s
Machine Gun ID	2
Main Gun ID	1
munition	ammo_sh_120_hesh_l31
surface	impacteffectsrock
Azimuth Trim	-0.8 deg
Elevation Trim	0 deg

On the right side, there are several buttons and controls:

- Refresh button (with a circular icon)
- Auto Refresh checkbox
- Default All button
- Rebuild All button

At the bottom left, there is a note: **default= 0**.

Time to launch VBS-IG from the Studio:



When IG is loaded and in standby mode, compile the vsTASKER database and run it.

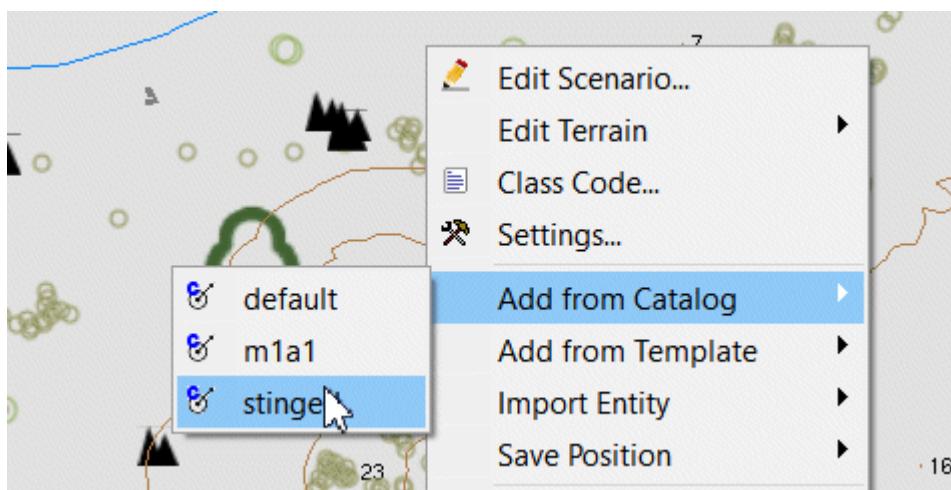
Click on the **m1a1** entity and use the **left** gamepad stick to turn the camera around (left and right triggers to zoom in and out).



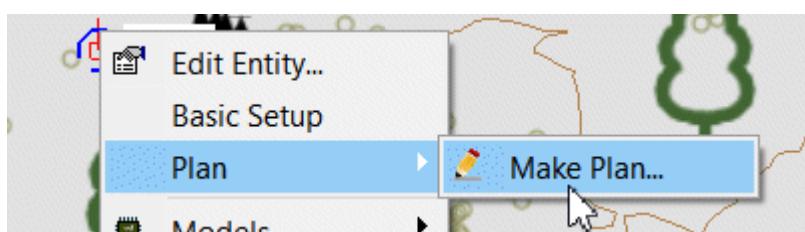
*If the camera FoV is not correct, you can reset it with the thumb pad up selector.*



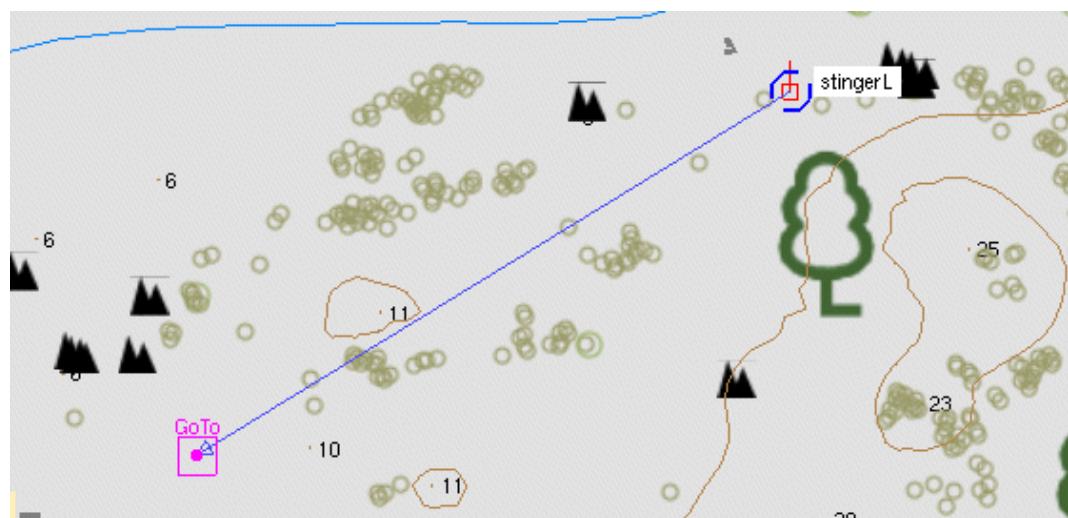
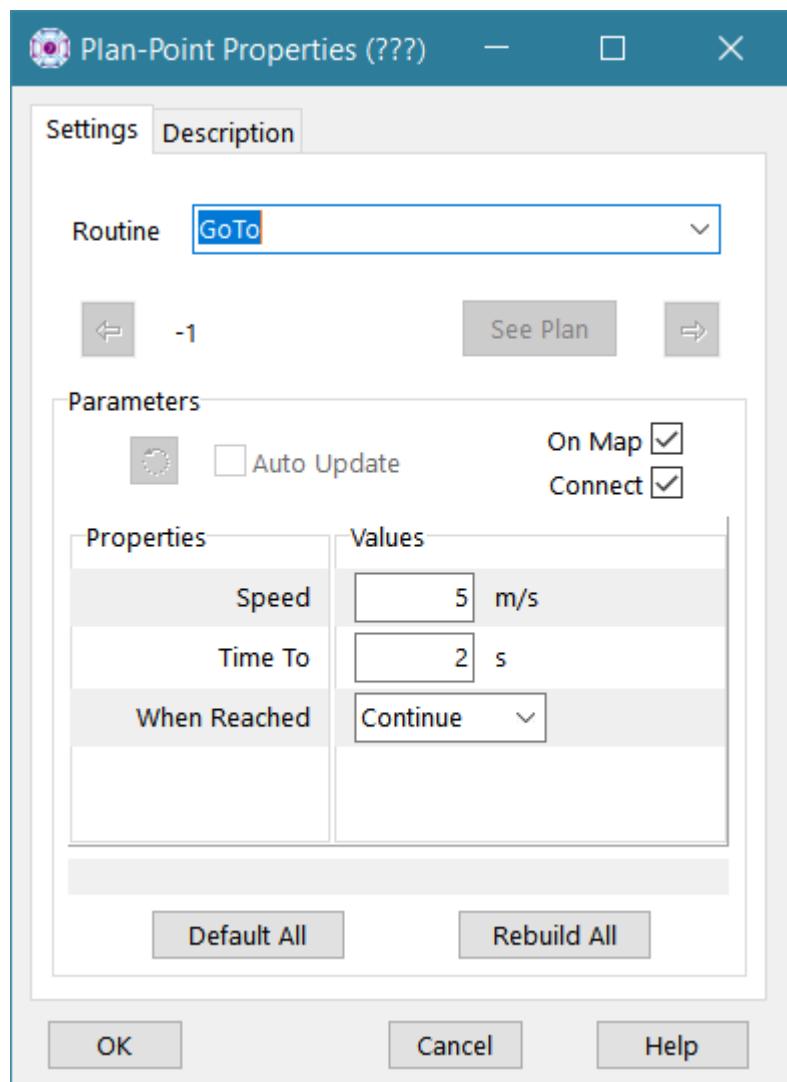
Now, let's add the Stinger from the Catalog list:



and let's give it a simple plan to move from one area behind the hill to the other side so that it becomes visible to the tank:



## Adding Entities



Compile and check again the simulation:

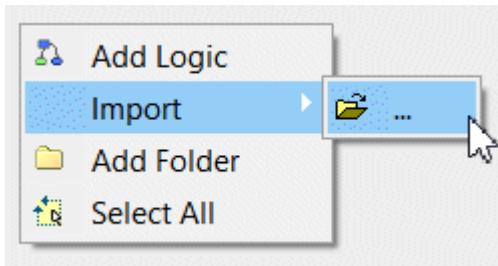


## Setting up the Tank

# Setting up the Tank

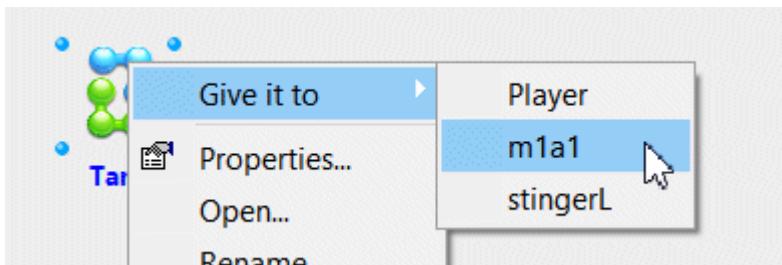
We want our tank to be able to shoot the main gun manually and automatically. We will import a predefined [Logic](#) (from the *Shared* library of stuff) which does that.

Select **Environment::Logics** then import a new [Logic](#) with a left click on the background:



and in folder [Shared/VBS-IG](#), select [Tank.lgk](#)

Once imported, give it to our **m1a1** tank (select the [Logic](#), right-click, **give it to...**) :



If we open the [Tank Logic](#), we can find three interesting parts inside the **Control Group**:

**CameraCtrl**: not used here but already set up to allow the camera to be positioned inside the tank and being aligned with the main gun and turret.

**TurretCtrl**: listens to the [Player](#) gamepad (only when entity is focused and activated) and orients the turret heading and main gun elevation according to the gamepad right stick. Right button fire the main gun.

**EngineCtrl**: same as *TurretCtrl* but use the gamepad right stick to accelerate and decelerate (up and down) or steer (right and left)



*Use gamepad A (green) button to switch from TurretCtrl to EngineCtrl when focused is on the tank.*

### **Setting up the Tank**

Now, recompile and run. Once focus, use the gamepad to move the turret and the camera to remain positioned behind. Try to shoot against the little hill.

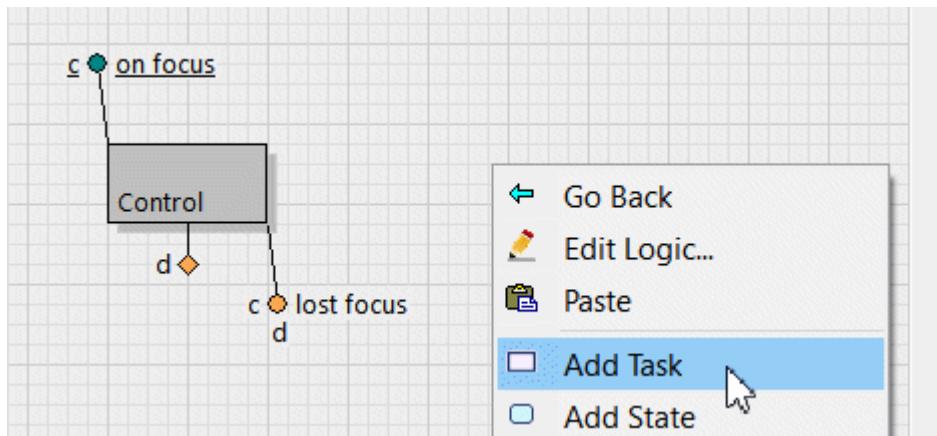


## Detect and Fire

# Detect and Fire

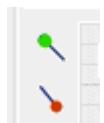
Let's create a basic task which will use the line of sight capability of the VBS-IG so that to automatically aim and fire at target whenever this one is visible from the tank. For that we will open the Tank Logic and add some objects for this new behavior.

In the **Tank** Logic, let's add a new **Task** (right-click on the logic background):



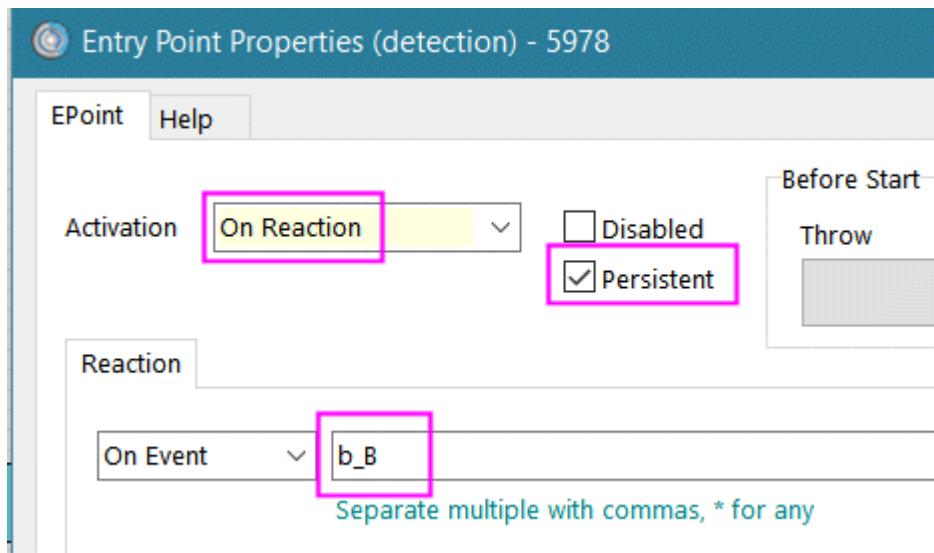
## • Detect

Let's rename it: **detection**.



Then let's add one **Entry Point** and one **Exit Point**. Click on the green needle then click on the Task to add the **EPoint** then click on the red needle then click on the Task to add the **XPoint**.

Now, double-click each of them and add the **b\_B** event as the reaction. This event is triggered by the gamepad when the B (red) button is depressed. Do not forget to make the EPoint persistent so that to reactivate it when the task is terminated:



Now, open the **detection** task and add the following code in each section:

#### Declaration:

```
private:
    int req, req_id, resp;
    Entity* target;
```

#### Initialization:

```
case RESET: {
    req = resp = 0;
    target = S:findEntity("stingerL");
```

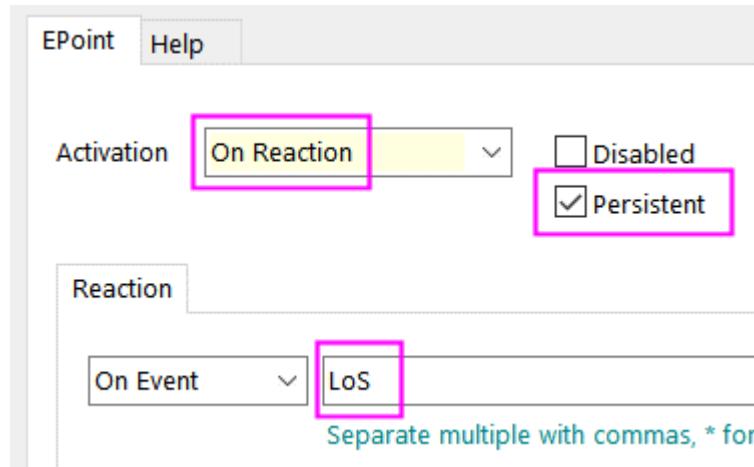
#### Runtime:

```
if (!req) { // ask for a line of sight
    req_id = RANDOM(1,100);
    cigi.LoSSegmentReq->request(ent(), target, req_id);
    resp = 0;
    req = 1;
}
else { // wait for response
    if (cigi.RespLoS->hasResponse(req_id)) {
        if (cigi.RespLoS->isVisible(target, req_id)) {
            // we have line of sight !!! Inform the Logic
            logk()->raiseEvent("LoS", target);
        }
        req = 0; // ready for a new request
    }
}
```

## Detect and Fire

### • Fire

Create a new Task named **autofire** and add a new **EPoint** with the following settings:



Now, open the **autofire** task and add the following code in each section:

#### Declaration:

```
private:  
    Entity* target;  
    VBS_MainGun* mg;
```

#### Initialization:

```
case RESET: {  
    target = (Entity*) edata();  
    mg = E:findVBS_MainGun();  
    mg->aimAt(target);
```

#### Runtime:

```
if (mg->ready()) {  
    // fire  
    mg->fireMainGun();  
}
```

Compile and run.

Move the tank to a location where the **stinger** will eventually be seen, activate the detection with the **B** button and wait.

**Detect and Fire**

## **VBS-3**

Based on the ASI interface, vsTASKER will send VBS script commands and will receive update data for all entities running in the VBS simulation for synchronization. It is then possible to let vsTASKER handle some parts of a scenario while leaving VBS handling the platform dynamics, the line of sight, or the exchange fire and damages. vsTASKER will enable and disable the AI according to situations and time. The end user will gain flexibility and control while building a specific scenario.

Using vsTASKER, all the scripts commands and functions of VBS can be encapsulated into actions, tasks, logic of vsTASKER and attached to any scenario entity on a simple mouse click. When running the simulation, logic can be visually monitored in real-time.

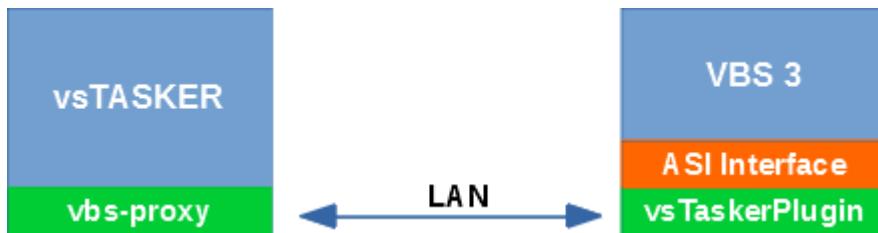
Because vsTASKER simulation engine and the ASI plugin are disconnected, the scenario development is quick: every time vsTASKER SIM starts, VBS creates locally all platforms and units defined in the scenario, then wait for commands to be unfold through the ASI interface. When vsTASKER stops, all VBS platforms and units are removed.



*Library only accessible to Defense version*

# Concept

The scheme behind this integration is to decouple the IG from the simulation engine, to even allow each to run on separate computers.



vsTASKER is using a specific **vbs-proxy** component which will package messages send by any places in the source code:

```

DTcommand cmd;

sprintf(cmd.str, "%s setManualControlled true;", E:vbs->var);
S:vbs_proxy->sendMessage(&cmd);

```

This suppose the understanding of the script commands of VBS3.

- **vsTaskerPlugin**

VBS allows a third party to communicate with the internal engine from a DLL using a scripting mechanism called ASI.

Unfortunately, the DLL is called at a low rate and the script processing takes quite a lot of time. Because both the ASI and the IG are inside the same thread, the impact on the IG frame rate is noticeable. Also, because the DLL is called at a low frequency, syncing the vsTASKER simulation engine with proper position of the VBS entities is not very efficient, even with extrapolation (VBS speeds are not what we expect).

The **vsTaskerPlugin** gathers all vsTASKER entities in a local list and tries to maintain a consistency with the VBS engine. When necessary, it updates vsTASKER with messages sent through the LAN to **vbs-proxy** component.

The communication with VBS core is done using the only provided function:

```

typedef int (WINAPI * ExecuteCommandType)(const char *command, char *result, int
resultLength);

```

## Concept

which processes the command and returns a result. This processing is CPU intensive and slows down the frame rate.



*Using vsTASKER scenario jointly with IA in VBS can become problematic when numerous entities are engaged because of the low rate syncing of entities. vsTASKER ends up with a situation which does not reflect in time the VBS situation, with reaction times which are not acceptable. It is better to switch OFF the VBS AI and force the position of entities.*

The plugin must be compiled prior to be loaded by VBS.

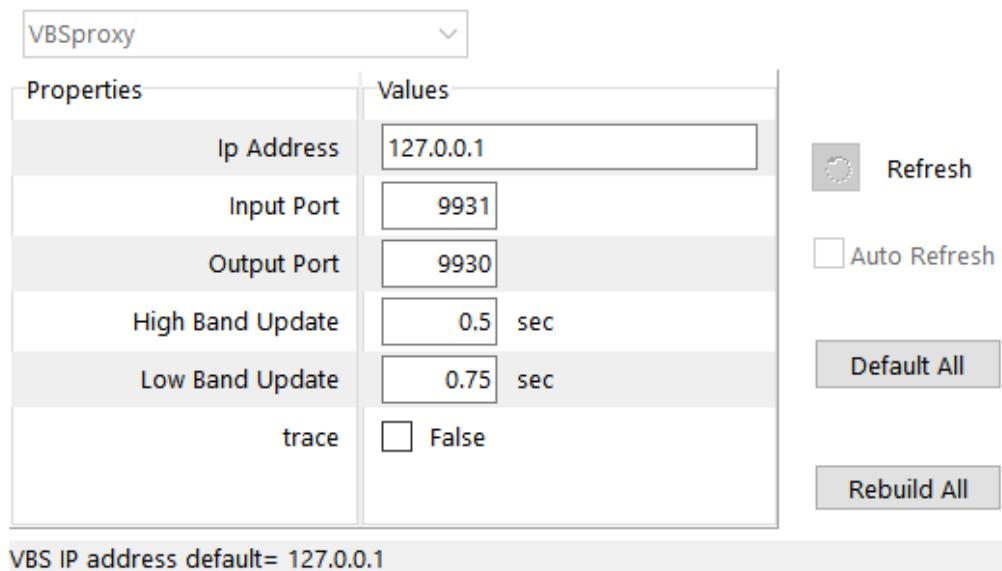
Make sure that you have set the environment variable: **BSIM\_DIR** to point to Bohemia directory (below VBS)

Open the solution in [/Runtime/VBS-3/Plugin](#) (vc100 or vc140) and generate the DLL in x64 bits. It should be stored in **\$(BISIM\_DIR)/VBS3/Plugins64**

## • vbs-proxy

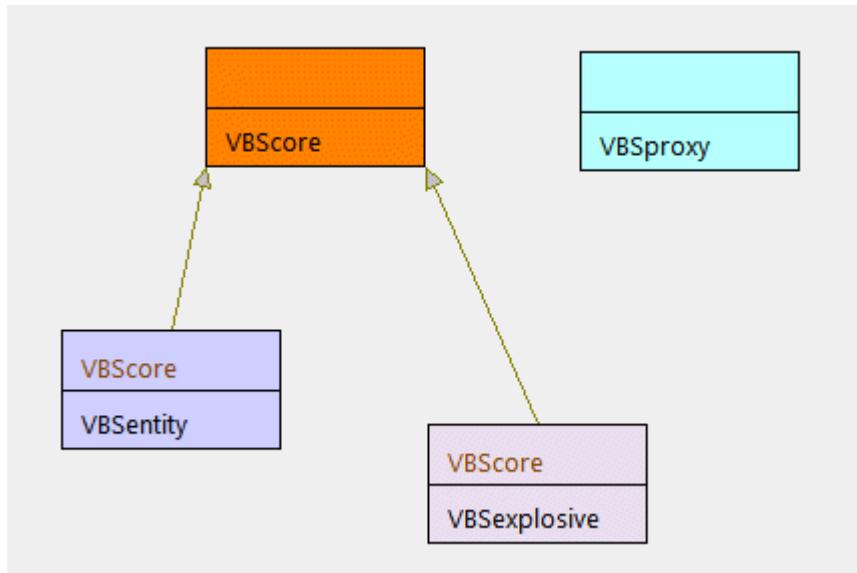
This component must be attached to the Scenario player and setup according to the [vsTaskerPlugin](#) static variable:

```
#define IN_PORT 9930 // vsTASKER -> VBS
#define OUT_PORT 9931 // VBS -> vsTASKER
#define CLIENT_IP "127.0.0.1" // Where to access vsTASKER
```



# Adding Entities

In vsTASKER, any entity which must be reflected in VBS using the ASI mechanism should have the [VBSEntity](#) component attached. Explosives like IED or mines should have the [VBSexplosive](#) component attached. The Development manual will explain the various components to use under ASI.



The component must be properly setup (here again, refer to the Development guide)

For the [VBScore](#) definition of a single soldier, select **Unit** then in the [Class Name](#), put the full VBS name as it can be retrieved from the VBS GUI itself. Specify the optional [Group](#) and most importantly, the unique name of the VBS object counterpart. All the commands sent to VBS relatively to this soldier will identify it using this variable name (**soldier**).

## Adding Entities

The screenshot shows the configuration dialog for a VBScore entity. The entity name is "VBScore". The properties listed are:

Properties	Values
Category	Unit
Class Name	VBS2_US_PMC_Rifleman_W_M4A1
Group	red
Variable	soldier

Buttons on the right include Refresh, Auto Refresh (unchecked), Default All, and Rebuild All.

Text at the bottom: default= Waypoint

Then, for the [VBSentity](#) definition:

The screenshot shows the configuration dialog for a VBSentity entity. The entity name is "VBSentity". The properties listed are:

Properties	Values
Drive Mode	Waypoint
Update Mode	PosHdgSpd
Update Band	Medium

Buttons on the right include Refresh, Auto Refresh (unchecked), Default All, and Rebuild All.

Text at the bottom: default= Waypoint

**Drive Mode** is set to **Waypoint** because the soldier will move according to a plan. The **Update Mode** will request from the [vsTaskerPlugin](#) position, heading and speed. **Medium** update frequency is enough as the soldier position on vsTASKER is not critical.

Now, compile and start the simulation.

The component will connect to the Player [vbs-proxy](#) component and automatically create the entity **soldier** (unit) in VBS.

If the soldier has a plan attached to it (with [VBSwaypoint](#) routines), then it will start moving on VBS and position will be updated on vsTASKER.

## **Adding Entities**

## Simple Logic

# Simple Logic

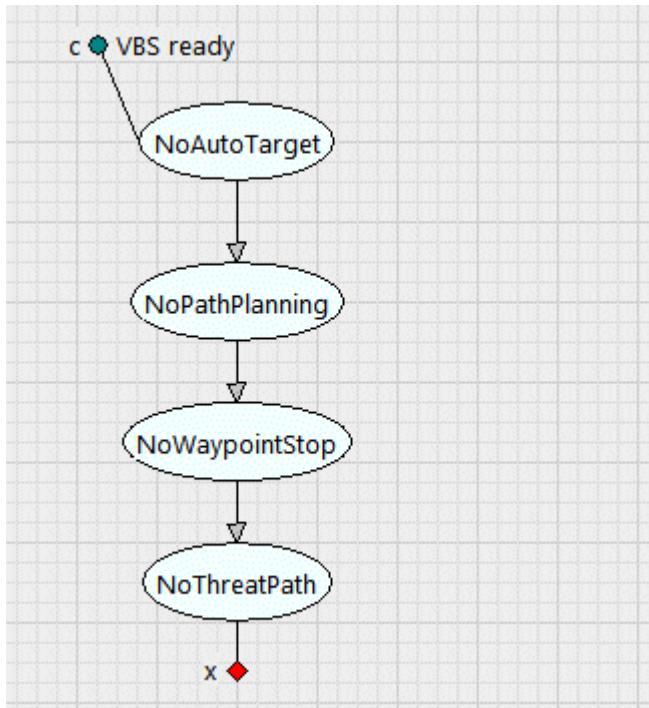
Once an entity (unit or vehicle) has been created on VBS by vsTASKER, sending instruction from a logic is quite simple.

It would be the same as using a script on VBS with much more ease.

VBSproxy component is retrieved globally using:

```
VBSproxy* proxy = (VBSproxy*) P:findComponent("VBSproxy");
```

- **Disable AI**



No Auto Target:

```
DTcommand cmd;
sprintf(cmd.str, "%s disableAI \\\"AUTOTARGET\\\"", E:vbs->var);
proxy->sendMessage(&cmd);
```

No Path Planning:

```
VBSproxy* proxy = (VBSproxy*) P:findComponent("VBSproxy");
DTcommand cmd;

if (E:vbs->isVehicle()) {
    sprintf(cmd.str, "(driver %s) disableAI \\\"PATHPLAN\\\"", E:vbs->var);
    proxy->sendMessage(&cmd);
```

```
}
```

### No Waypoint Stop:

```
DTcommand cmd;

if (E:vbs->isVehicle()) sprintf(cmd.str, "(driver %s) disableAI \\\"WAYPOINT_STOP\\\"", E:vbs->var);
else if (E:vbs->isUnit()) sprintf(cmd.str, "%s disableAI \\\"WAYPOINT_STOP\\\"", E:vbs->var);
proxy->sendMessage(&cmd);
```

### No Threat Path:

```
DTcommand cmd;

if (E:vbs->isVehicle()) sprintf(cmd.str, "(driver %s) disableAI \\\"THREAT_PATH\\\"", E:vbs->var);
else if (E:vbs->isUnit()) sprintf(cmd.str, "%s disableAI \\\"THREAT_PATH\\\"", E:vbs->var);
proxy->sendMessage(&cmd);
```

## • Manual Control

### Speed Limit:

```
DTcommand cmd;

if (E:vbs->isVehicle()) {
    sprintf(cmd.str, "%s setMaxSpeedLimit 90;", E:vbs->var);
    proxy->sendMessage(&cmd);
}
```

### Steering:

```
DTcommand cmd;
sprintf(cmd.str, "%s setTurnWanted %.3f; %s setThrustWanted %.3f;",
        E:vbs->var, turn_factor, E:vbs->var, speed_factor);
proxy->sendMessage(&cmd);
```

## • Explode

```
DTcommand cmd;

Entity* ied = (Entity*) S:findEntity(E:attached_ied);
```

## Simple Logic

```
if (ied) {  
    sprintf(cmd.str, "%s setDamage 1;", ied->vbs->var);  
    proxy->sendMessage(&cmd);  
}
```

## Titan Vanguard

Based on the C++ API of Titan, vsTASKER will encapsulate its simulation engine into a DLL to be loaded by Titan.

The DLL will be called at every cycle and vsTASKER RTC will be acting as the game maker.

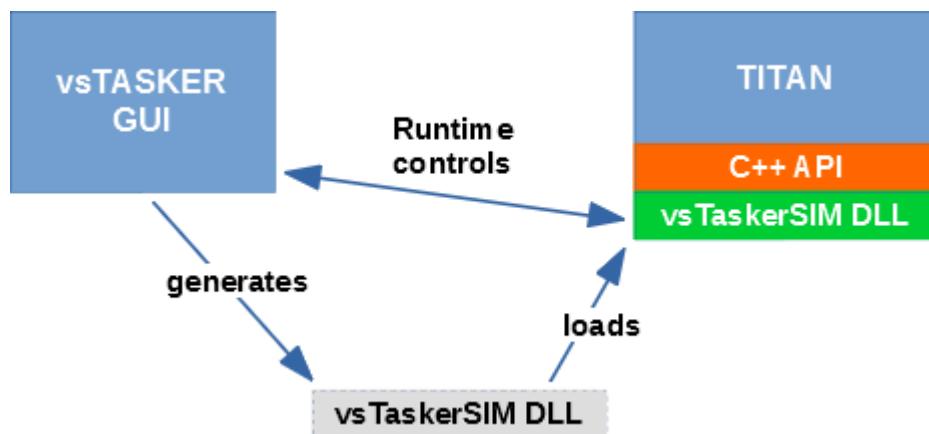


*Specific license is mandatory. Contact VirtualSim if you need the CIGI module.*

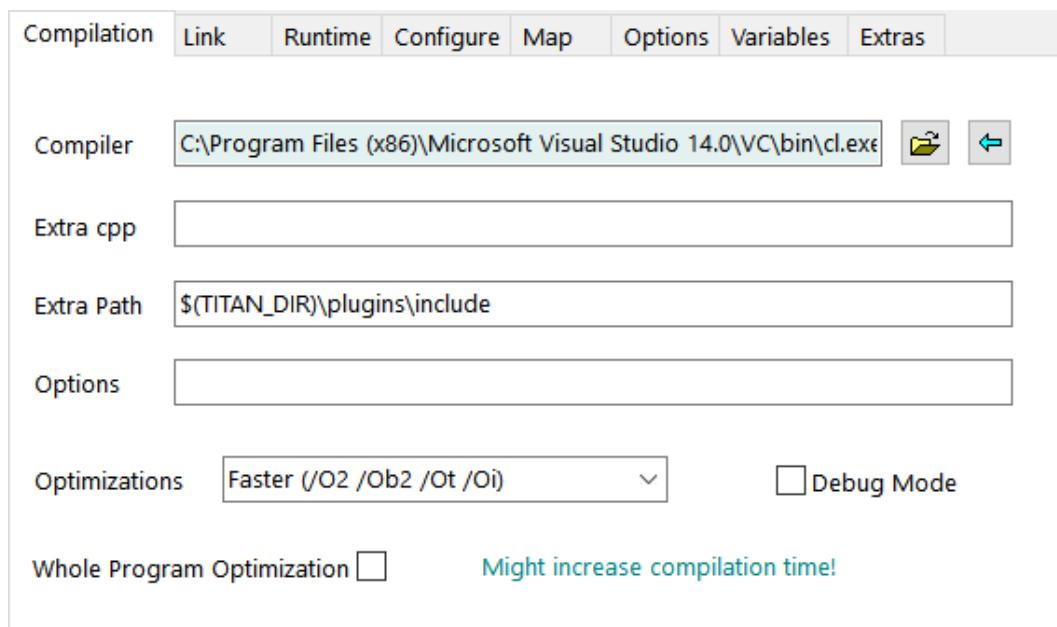
## Concept

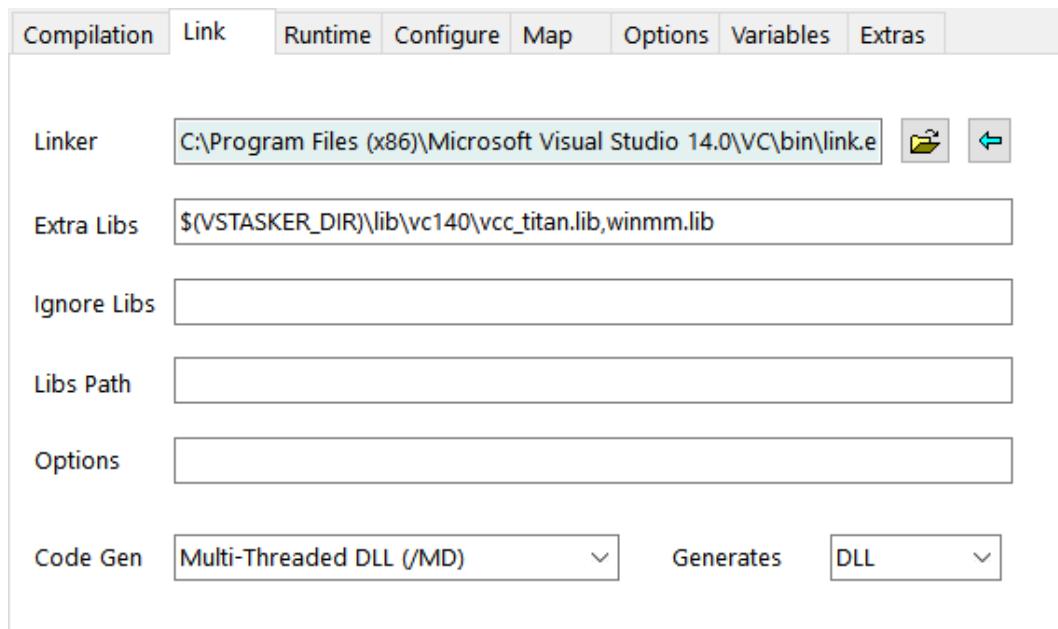
# Concept

To integrate with Titan, vsTASKER generates a simulation engine as a DLL.



For that, the database settings must be done as below:





The [Titan](#) viewer must also be used.

## • Using Visual Studio

If using the debug solution, open the [/Runtime/Titan/vc140/vsTaskerTitan.sln](#)



*Make sure to define the environment variable `TITAN_DIR` pointing to the `/titan` product installation directory.*

## Scenario Setup

# Scenario Setup

- In **Database::Settings::Runtime**, make sure that **Directory** is set to: `$(TITAN_DIR)\plugins`. vsTASKER will generate a DLL to be loaded by Titan.

Make sure that **Database::Settings::Link** generates a DLL

- In **Classes::Global::Definitions**, add the following:

```
#include "titan/vt_titan.h"
```

The `vt_titan.h` (`vcc_titan.lib`), contains an API to simplify the coding of some Titan interactions. It will contain more and more functions by time.

- In **Classes::Global::Declaration**, add this line:

```
extern TitanRoot* titan_root;
```

The `titan_root` variable is setup by the Titan viewer code (`vt_runtime_titan.cpp`). It is defined in `vt_titan.h` and contains major pointers to the Titan core engine (interface).

- In **Classes::Global::Declaration**, setup the `titan_root` pointer:

```
case INIT: {
    titan_root = (TitanRoot*) udata();
} break;
```

As `titan_root` is declared external, the setup is not mandatory.

- In **Classes::Scenario::Declaration**, add the following public static method:

```
static float titanGetAlt(WCoord&, float offset =0);
```

and define it in **Classes::Scenario::Methods**:

```
// warning: this function is static!
float Scn::titanGetAlt(WCoord& pos, float offset)
{
```

```
WCoord lla = pos;
lla.convertToLLA();
float alt = vtTitan::getTerrainHeightAt(lla.lat, lla.lon);
return alt + offset;
}
```

You can decide to use this function in replacement of the default vsTASKER `getAlt()` or whenever you need it. If the scenario has a terrain elevation loaded, calling this function might not be necessary as it is CPU intensive.

- In **Classes::Entity::Declaration**, add the following public pointer:

```
TitanEntity* te;
```

and in **Initialization INIT** part, set it this way:

```
te = (TitanEntity*) find(TT_Component, "TitanEntity");
```

This component is mandatory for an entity to be created in Titan engine. See the **Development Guide** for more information on this component.



*The database template Titan contains already these settings.*

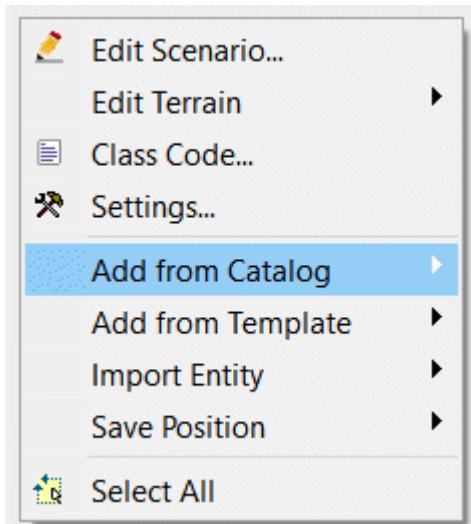
## **Adding Entities**

# **Adding Entities**

When adding entities to a scenario, two options must be chosen:

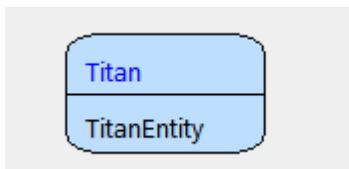
- 1- the entity is driven by vsTASKER**
- 2- the entity is driven by Titan.**

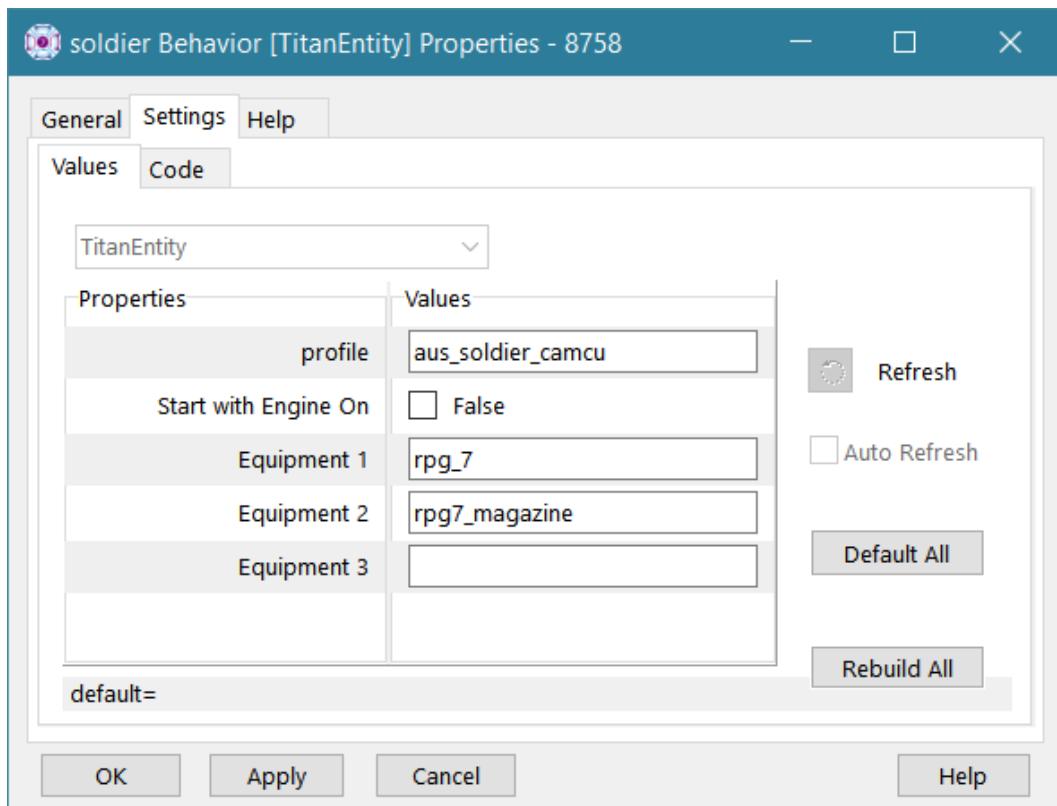
Add a basic or catalog entity to the scenario:



**Catalog**: predefined entities already setup to work with Titan, or **default**  
**Template** embedded entities (if any) attached to the loaded template (if any).  
**Import**, from file (shared directory), if any.

Once the entity is on the map, select it and go to the Model panel to add and setup the **TitanEntity** component:





The Profile value should be the type name defined on the Titan Entities window:



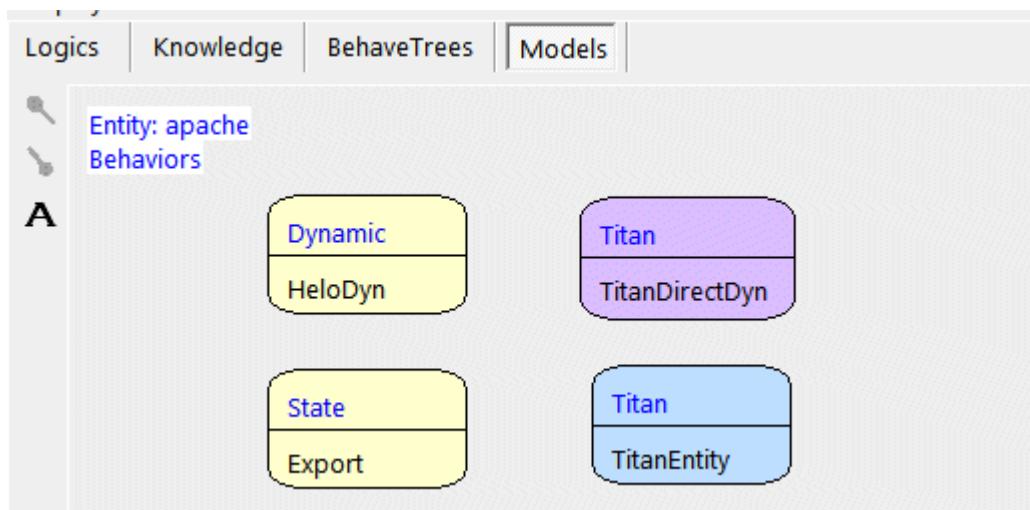
If the entity need equipments or weapons, list them with their Titan type name. Either refer to the documentation of Titan or find the name in the appropriate window or text field in Titan.

*For the moment, this part is the most difficult and frustrating as finding the name for an entity or equipment is not obvious.*

*Parse the `/titan/packages` directory to find some of them. Open the `entdef` file of each entity and look for the `entityKeyName` to use.*

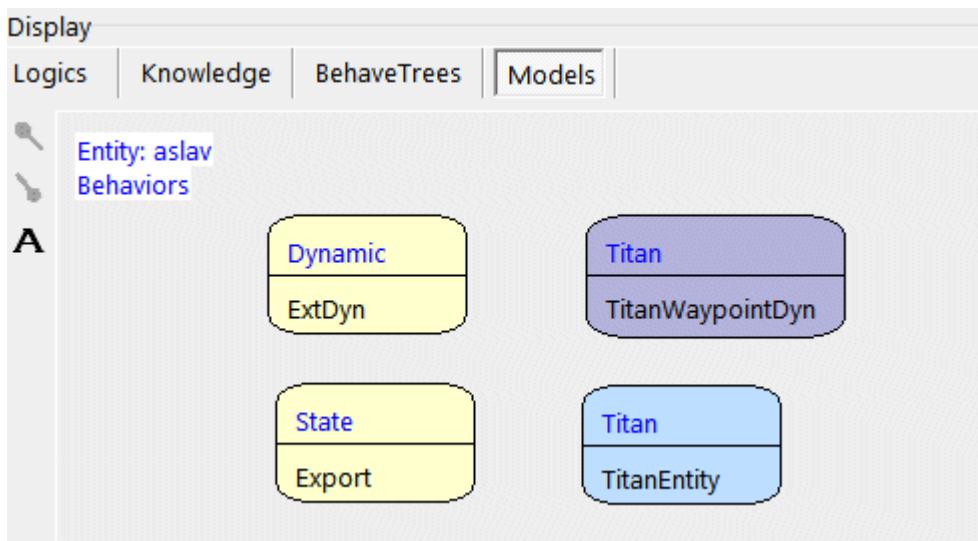
- **Driven by vsTASKER**

## Adding Entities



Because we are using a vsTASKER motion dynamic ([HeloDyn](#)), we need to add the [TitanDirectDyn](#) component (see the **Development Guide** for more information on it). This way, the [DirectDyn](#) component will update the position of the Titan entity in real-time.

## • Driven by Titan

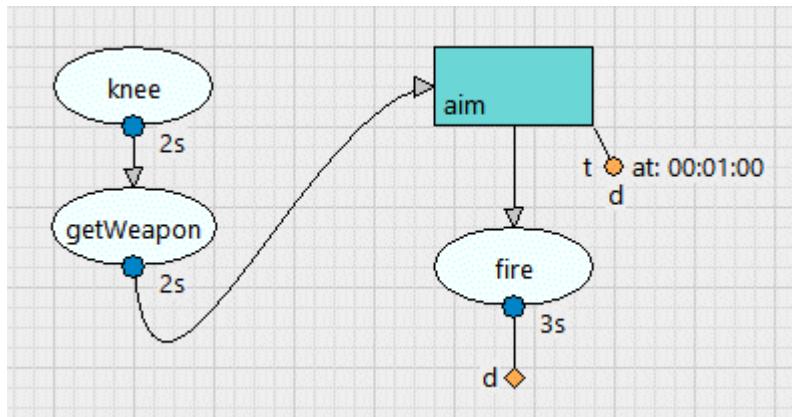


In case we want Titan dynamics to be used, we must add the [ExtDyn](#) component combined with the [TitanWaypointDyn](#) one. This way, vsTASKER will be driving entities by assigning waypoints and let Titan does the computation. The [TitanWaypointDyn](#) component will give instructions to Titan and update the position and attitude of the local entity in real-time, from Titan engine.

# Firing a Weapon

- Equipped soldier

Create a simple logic with the following actions:



Knee:

```
E:te->setPostureKneeling();
```

getWeapon:

```
using namespace titan::api2;

//shared_ptr< IEntity> vis = loadEquipment("vision");
shared_ptr< IEntity> rpg = ent()->te->getWeapon("rpg_7");
if (rpg) ent()->te->ent_character->setEquippedWeapon(rpg->getWeaponTraits());
else ent()->inform("Cannot get RPG-7");
```

aim:

```
ent()->te->ent_character-
>setWeaponPosture(titan::api2::WeaponPostureAiming);
```

fire:

```
using namespace titan::api2;
shared_ptr< IWeaponTraits> rpg = ent()->te->titan_ent->getActiveWeapon();
if (rpg) rpg->fireSingle();
```

## Firing a Weapon

### • Main Gun

```
TitanEntity* te = ent()->findTitanEntity();
shared_ptr<titan::api2::IWeaponTraits> weapon =
vtTitan::getWeapon( te->titan_ent, 1);
if ( weapon ) weapon->fireSingle();
```

# VegaPrime

VegaPrime viewer facilitates the integration of vsTASKER simulation engine with Presagis VegaPrime IG.

The Viewer will combine VegaPrime library with vsTASKER runtime libraries, outputting a unique (and possibly standalone) application.

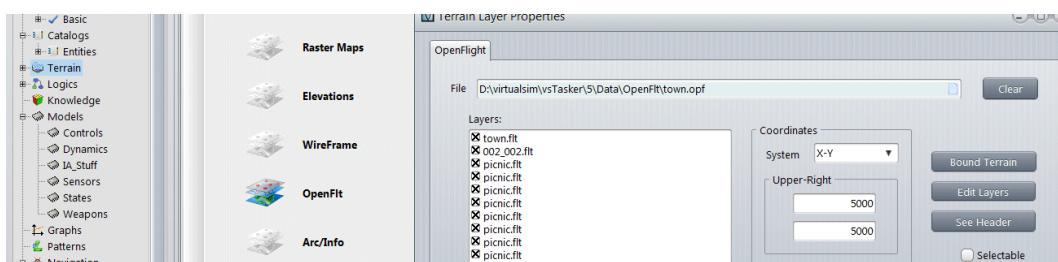
In this sample, we will use an existing VegaPrime terrain (town) to put a tank on the scenario and control it.

Create a new Database from the VegaPrime template.

Select **Basic** scenario template.

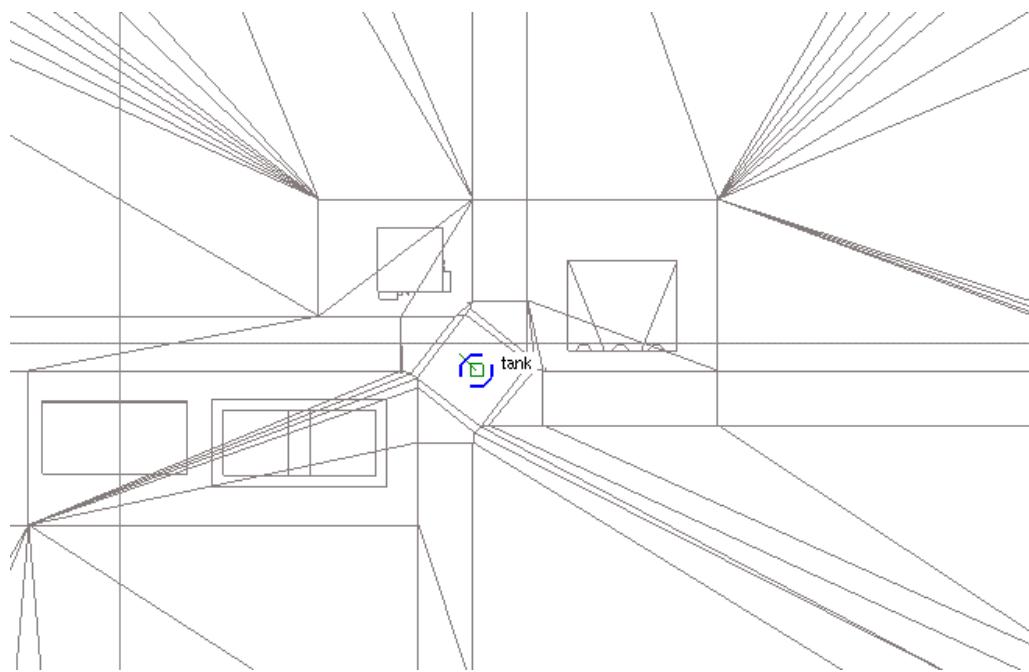
Then **save** it under the name **test\_vprime**.

Load the OpenFlight **town.opf** terrain file, in **Environment::Terrain::OpenFlt:**

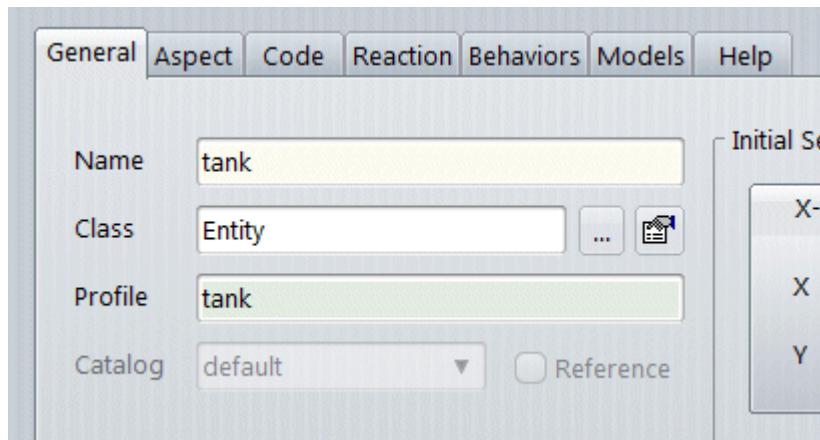


On the terrain map, drop the **default** entity in the middle of the road intersection:

## VegaPrime



On its property window, rename the entity "tank" and also, put "tank" on its **Profile**. This profile identification will be used at runtime to load from the **acf** file the correct 3D model (see below).



It is convenient to use the **Profile** text field but this is not mandatory. The way the template entity uses it at initialization time is written in the entity **Code Initialization** panel:

Entity Properties (tank) - 91 slot:0 (Basic)

General Aspect Code Reaction Behaviors Models Help

Initialization Runtime Display

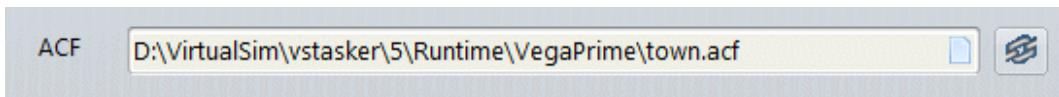
```
// Instance Entity initialization phase
// function MData* udata() if not NULL can be used.

switch (_phase) {

    case INIT: {
        vp_obj = vpobject::find(db->getProfile());
    } // no break, RESET will also be processed

    case RESET: {
```

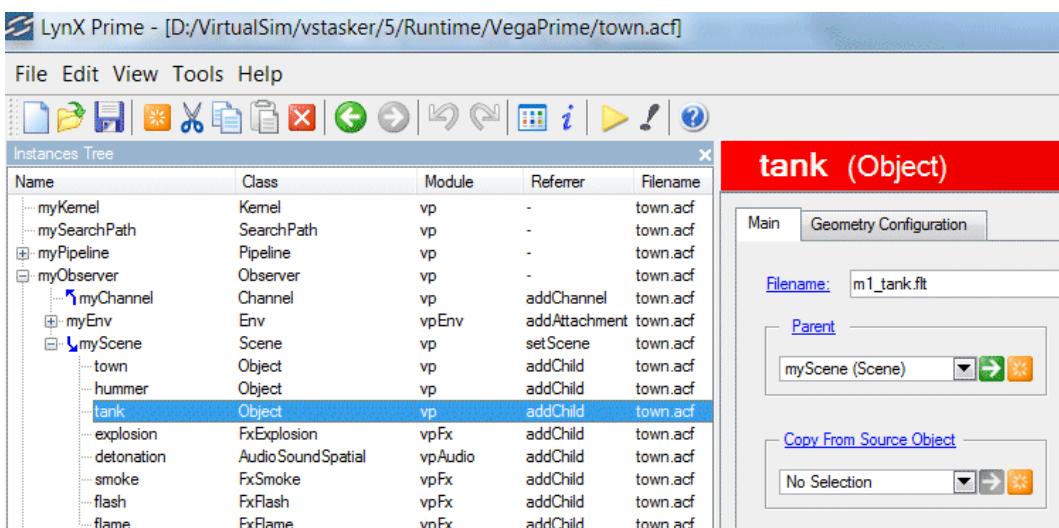
Now, add a VegaPrime **Player** and select, as an **acf** file, the provided **town.acf** located in **/runtime/vegaprime**.



Use button to open **Presagis/Vega\_Prime\_3/bin/LynxPrime.exe**

You will notice that tank model has been defined.

If you need to add or prepare mode 3D models or effects, just add them in the **acf** file and use them from inside vsTASKER:



Save and build the simulation engine then run it.

## VegaPrime

You will get the tank put at the exact location on the 3D view.

Use the mouse to control the camera and the entity hook window to make it move on the scene.



**NOTE** For a better clamping of the tank, change its dynamic model from *LinearDyn* to *LandDyn* and select *Clamped* to true.

## **CIGI Simple**

This example will simply use Pendleton military airbase and one F16 which will taxi, takeoff, land, taxi and park, automatically.

You will need to start VegaPrime Lynx and open the cgi\_simple.acf file in /runtime/vegaprime. It is already configured with the cgi\_simple.mft file located in /runtime/CIGI/settings/vegaprime. You will have to edit this one to match the installation path of your VegaPrime. The file must be found for the 3D model to be displayed. Same thing for pendleton database. Make sure that the path is correct.

# STK

This tutorial has been tested under STK 9.

Migration to later versions of STK remains on the customer side for the moment. Contact support ([support@vstasker.com](mailto:support@vstasker.com)) if you have problems with this demo.



*Specific license is mandatory. Contact VirtualSim if you need this module.*

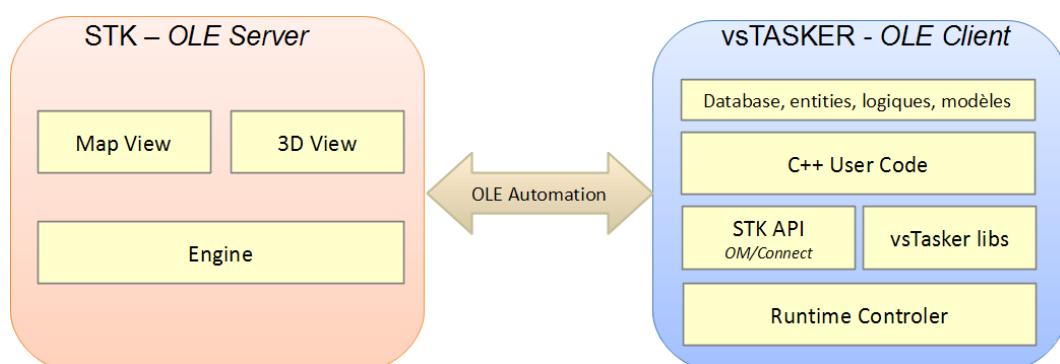
- **Principle**

Most of AGI's customers are using STK for space, aircraft, UAV and intelligence mission design and analysis.

It includes trajectories and attitudes tuning, multi-sensors coverage computation taking into account terrain masking or budget links assessment. All results are visualized in a real time performing 3D viewer.

Up to now, behaviors of STK entities were purely deterministic that could induce some restrictions in scenario design process when faced with real operations complexity.

To answer this challenge, STK can be interfaced with vsTASKER to bring Event-Driven Behavioring capacity into STK. In managing behaviors and logic transitions of all assets during scenario runs, the combined application STK/vsTASKER produces a more realistic simulation than ever, putting similar reactivity as real operational theater.



*STK software and license must be requested from a proper vendor (AGI), as vsTASKER does not provide them nor behave as a reseller or support line. Supported version goes up to v3.*

- **Automation**

Automation with STK is similar to automation with MS Office.

So, the following article about « [Office Automation Using Visual C++](#) » is a good entry point if you are not familiar with OLE Automation.

### **“Here are three basic ways you can use Automation: MFC, #import, and C/C++:**

With MFC, use the Visual C++ [ClassWizard](#) to generate "wrapper classes" from the Microsoft Office type libraries. These classes, as well as other MFC classes, such as `ColeVariant`, `ColeSafeArray`, `ColeException`, simplify the tasks of Automation. This method is usually recommended over the others, and most of the Microsoft Knowledge Base examples use MFC.

`#import`, a new directive that became available with Visual C++ 5.0, creates VC++ "smart pointers" from a specified type library. It is very powerful, but often not recommended because of reference-counting problems that typically occur when used with the Microsoft Office applications.

C/C++ Automation is much more difficult, but sometimes necessary to avoid overhead with MFC, or problems with `#import`. Basically, you work with such APIs as `CoCreateInstance()`, and COM interfaces such as `IDispatch` and `IUnknown`.

## **• Example using MFC**

First, use the "MFC Class from TypLib" wizard of VisualStudio to generate the header files of the STK classes.

Then, you can use the generated wrappers:

```
#include "CIAgStkObjectRoot.h"
// headers have been generated with VisualStudio using the "MFC
Class from TypLib Wizard"
...
CIAgStkObjectRoot* stkRoot;
...
stkRoot=new CIAgStkObjectRoot(m_stkUiApp.get_Personality2());
stkRoot->ExecuteCommand(myCmd);
```

## **• Example using smart pointers**

```
#import "C:\Program Files\AGI\STK 9\bin\AgSTKUtil.dll"
no_namespace
#import "C:\Program Files\AGI\STK 9\bin\AgVGT.dll"
no_namespace
```

## STK

```
#import "C:\\Program Files\\AGI\\STK 9\\bin\\AgSTKObjects.dll"
no_namespace
...
IAgStkObjectRootPtr stkRootPtr;
...
stkRootPtr= m_stkUiApp.get_Personality2();
stkRootPtr->ExecuteCommand(myCmd);
...
```

## • Example using COM interfaces

You can use the native COM API or use the ATL API which facilitates the usage of COM components.  
With ATL:

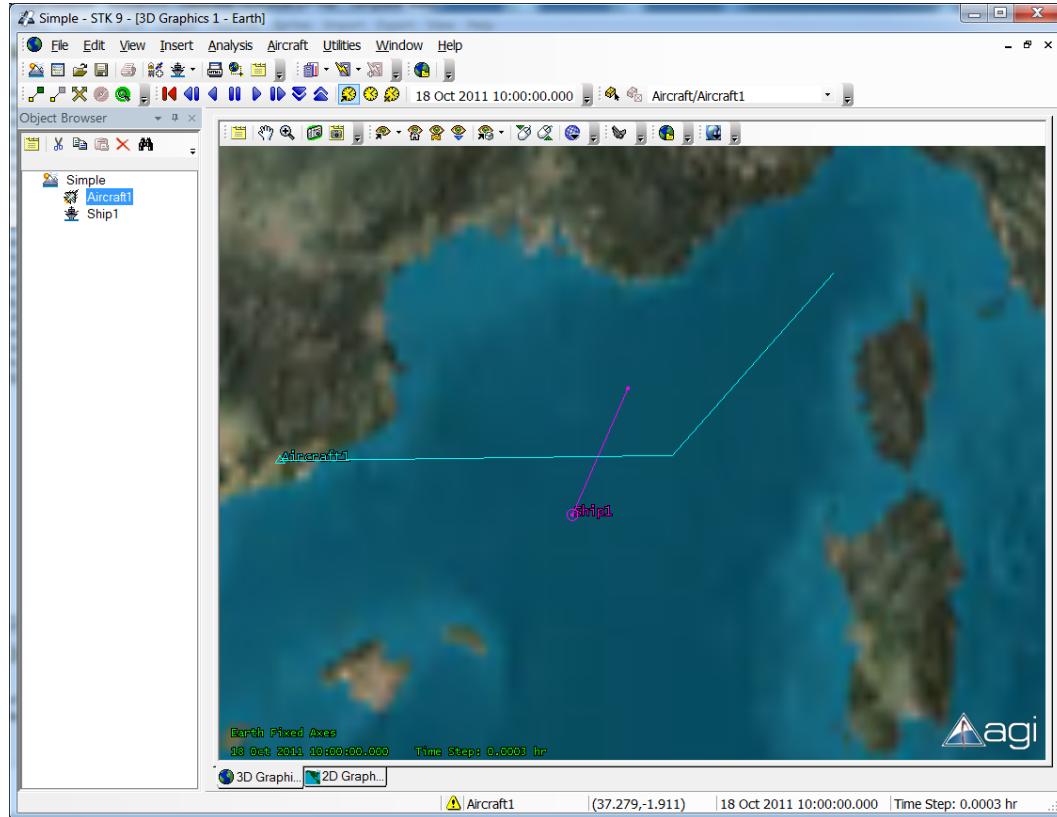
```
#include "Afxctl.h"
...
CComQIPtr<IAgStkObjectRoot> stkRootPtr;
...
stkRootPtr.CoCreateInstance(__uuidof(AgStkObjectRoot));
stkRootPtr->ExecuteCommand(myCmd);
...
```

**Simple**

# **Simple**

## Create STK Scenario

# Create STK Scenario



In this example, we will just create a basic simple scenario into STK.

We will create two entities:

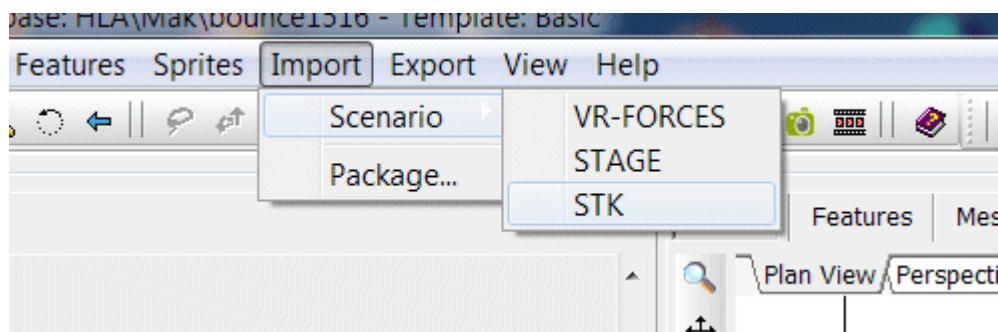
One [Aircraft1](#) (cyan) with a route like shown on the above screen copy.

One [Ship1](#) (magenta) with a route crossing the [Aircraft1](#) one.

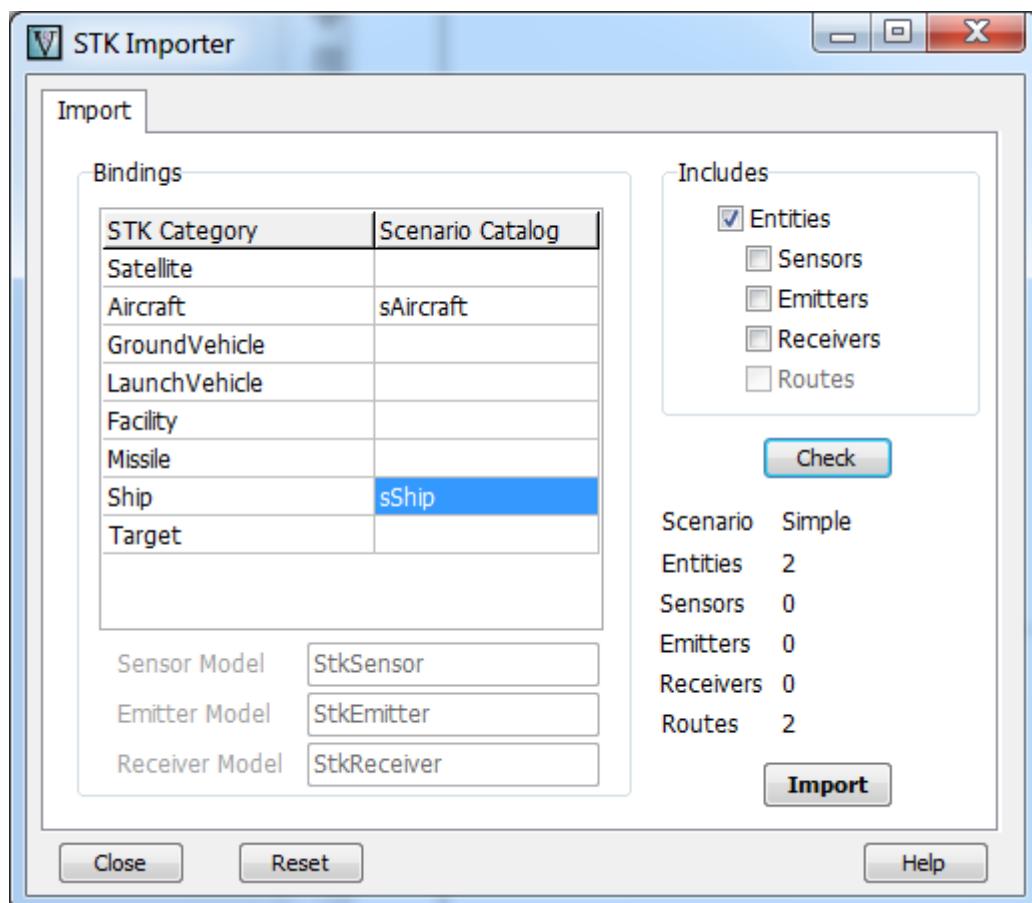
The idea is that [Aircraft1](#) and [Ship1](#) will meet, so that a sensor mounted on [Aircraft1](#) will be able to detect [Ship1](#).

# Importing Scenario

Now, we will import the STK scenario (created [here](#)) into vsTASKER. We need to get the STK Imported window using the following command:



to get the importer:

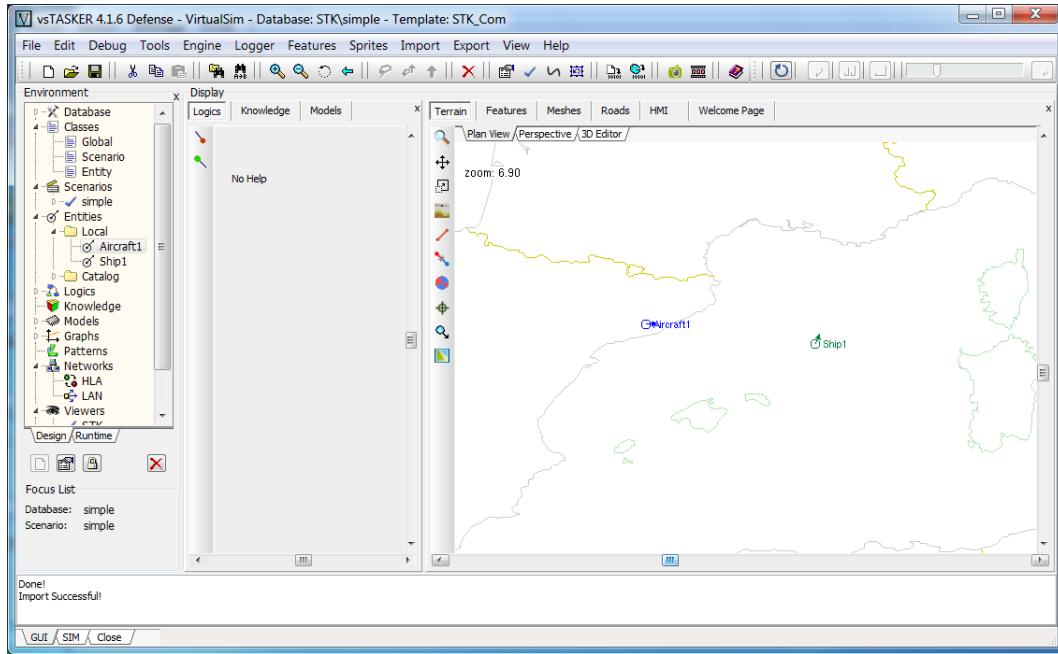


As we want to import the Aircraft and the Ship, we will only select the two Catalog entities to match the STK corresponding Category.

Then, we click the Import button.

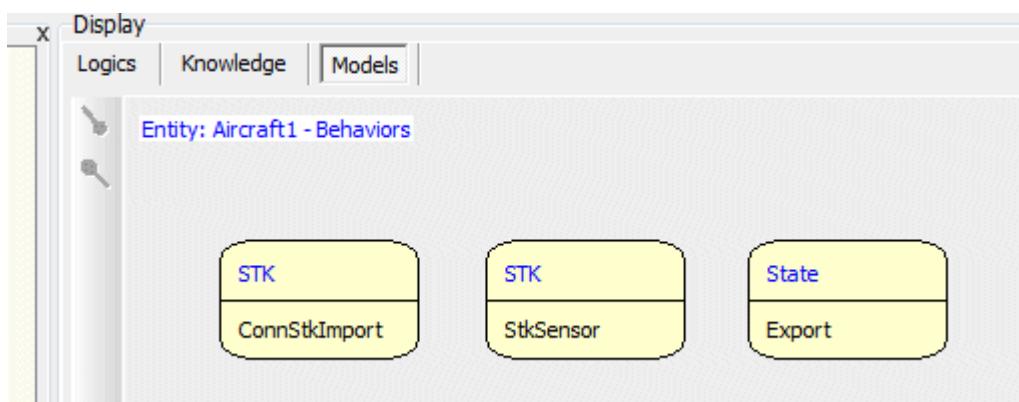
## Importing Scenario

(For detail explanations on the window, refer here).



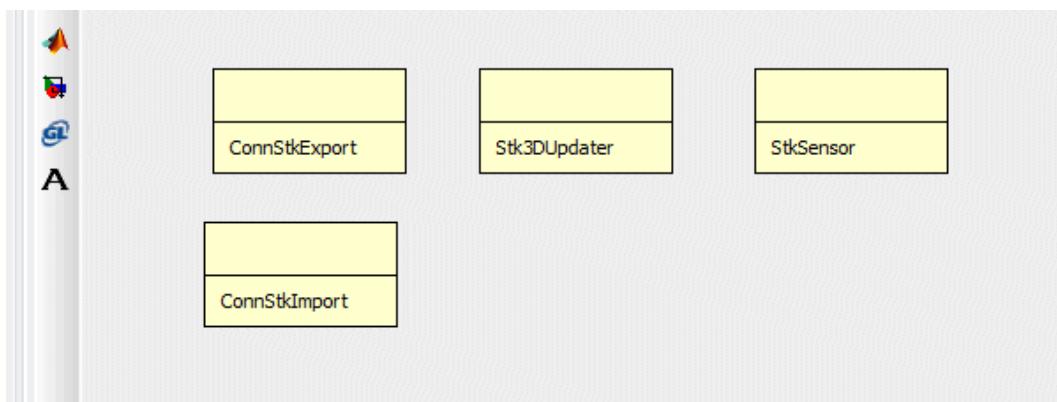
We can notice that the two entities have been created on vsTASKER map.  
We have not imported routes because STK will manage it anyway.

If you select [Aircraft1](#) or [Ship1](#), you will notice that they are using some STK specific Components.

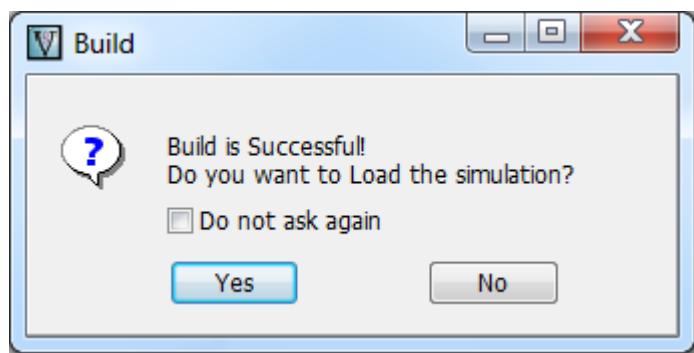


These Components are defined in the STK Model folder and already attached to predefined Catalog Entities to speed up the design process.

## Importing Scenario



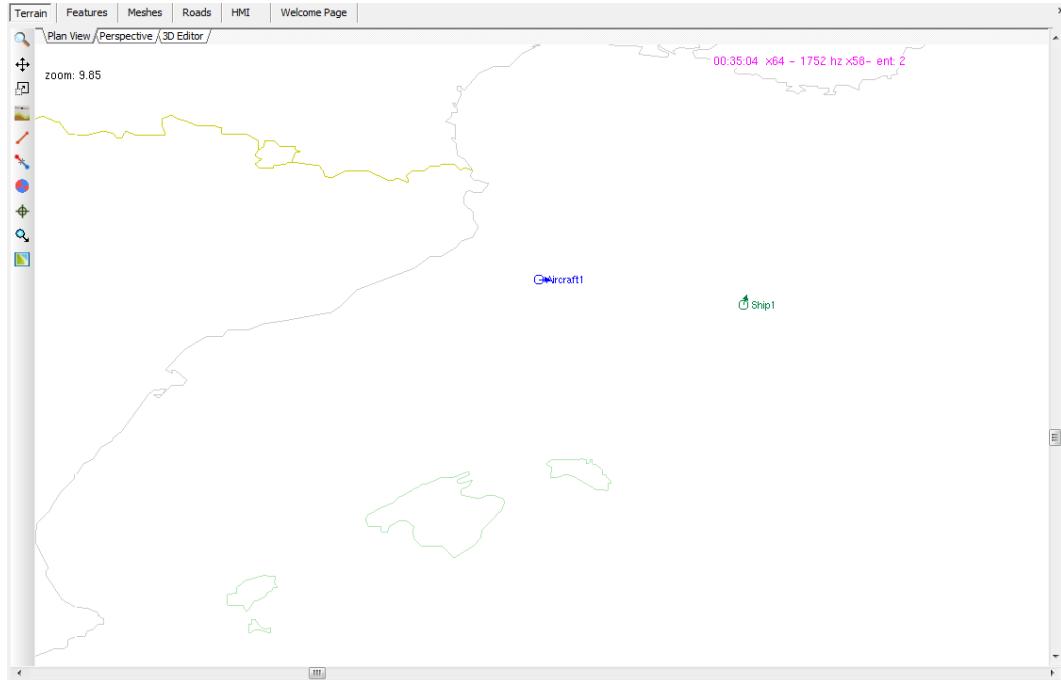
Now, we can just produce and compile the simulation engine.



and then run the simulation.

## Running the Simulation

# Running the Simulation



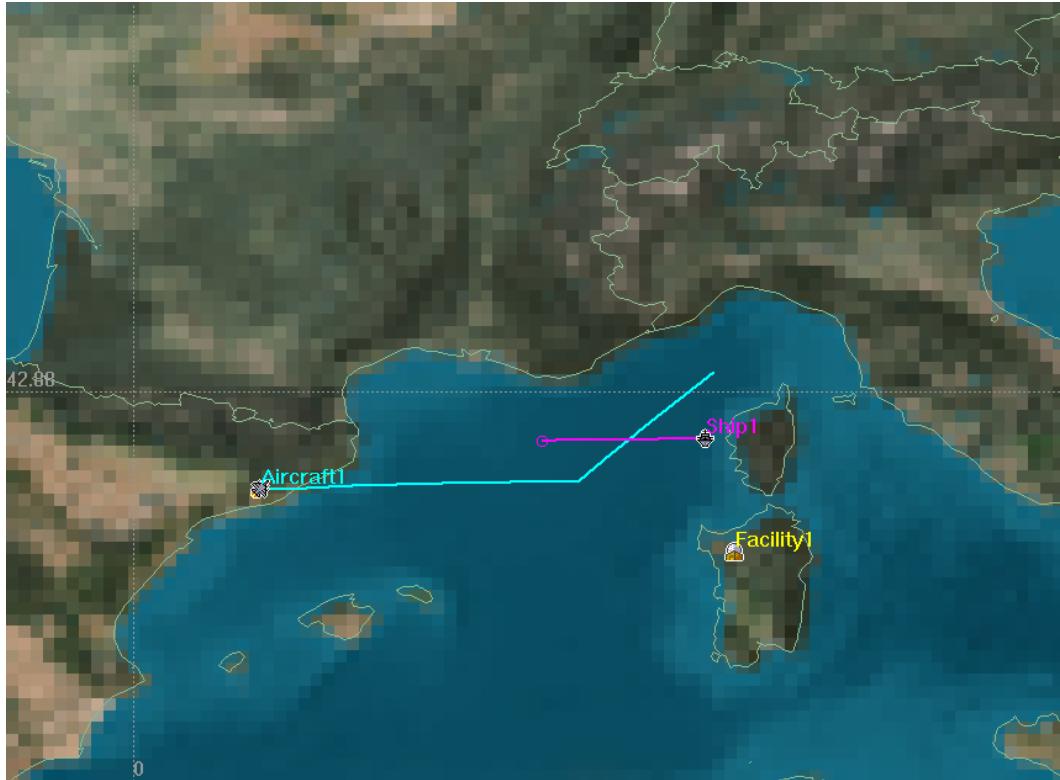
Because vsTASKER simulation engine controls the time, the simulation can run up to the maximum speed STK can response.

Every vsTASKER SIM Engine tick, STK will be called to update all entities handled by STK, at a frequency specified at the vsTASKER component level.

During the simulation run, both vsTASKER GUI and STK GUI can be used. Both applications are also synchronized and display the same situation.

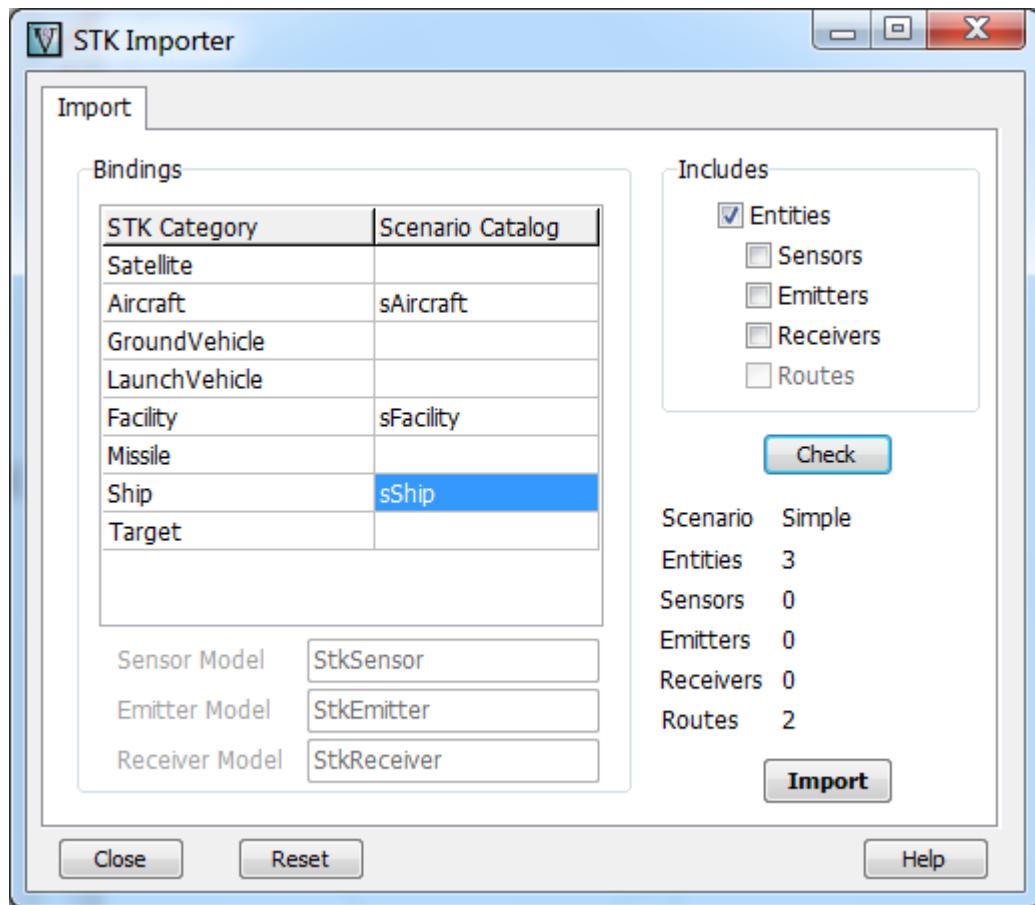
## Updating the Scenario

Now, let's change the position of the [Ship1](#) while insuring it will still meet [Aircraft1](#). Let's also add a [Facility1](#) northwest of Sardinia.



To update the vsTASKER scenario, just recall the STK Importer window, keep the bindings for Aircraft and Ship, add the Facility Catalog to match the STK Category and Import.

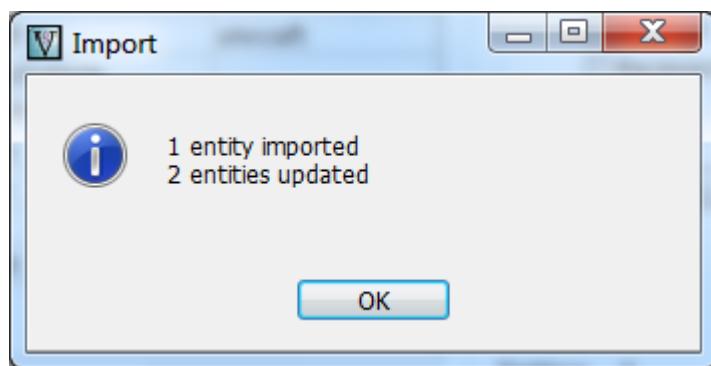
## Updating the Scenario



A message window will inform that 1 new entity ([Facility1](#)) has been imported and that the 2 existing ones ([Aircraft1](#) and [Ship1](#)) have been updated.

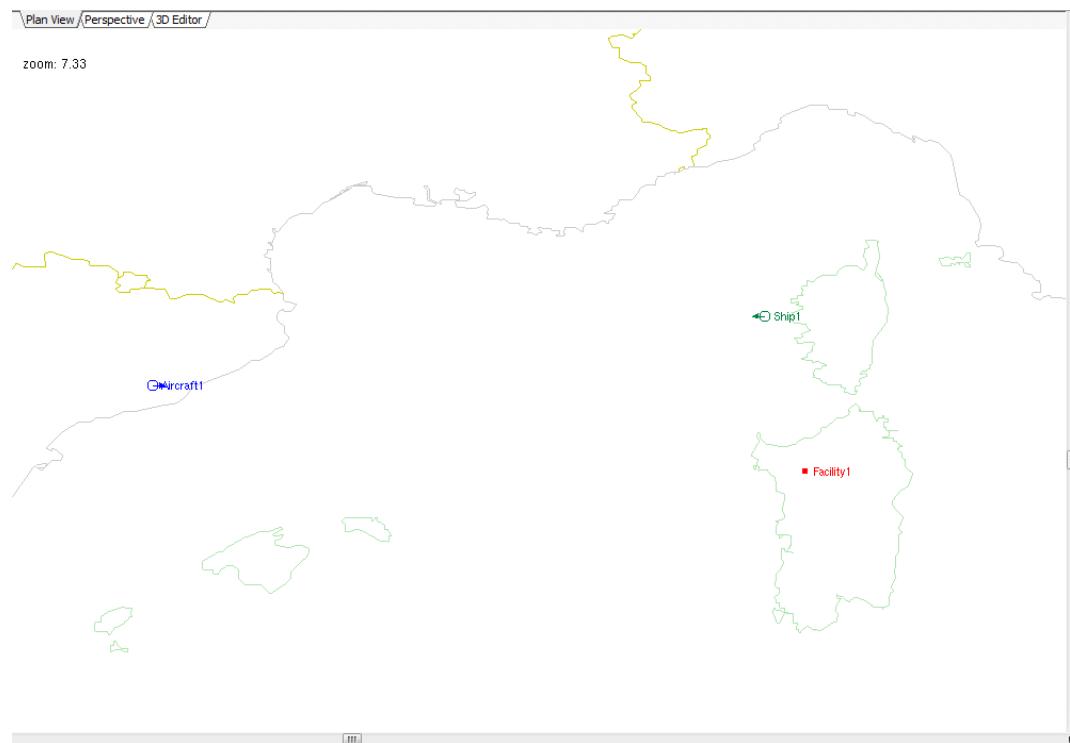
Of course, names are important.

If an imported STK entity is renamed, the orphan vsTASKER (old) one will be automatically removed.



Now, we can see the new [Facility1](#) entity in red located at the same exact position.

## Updating the Scenario



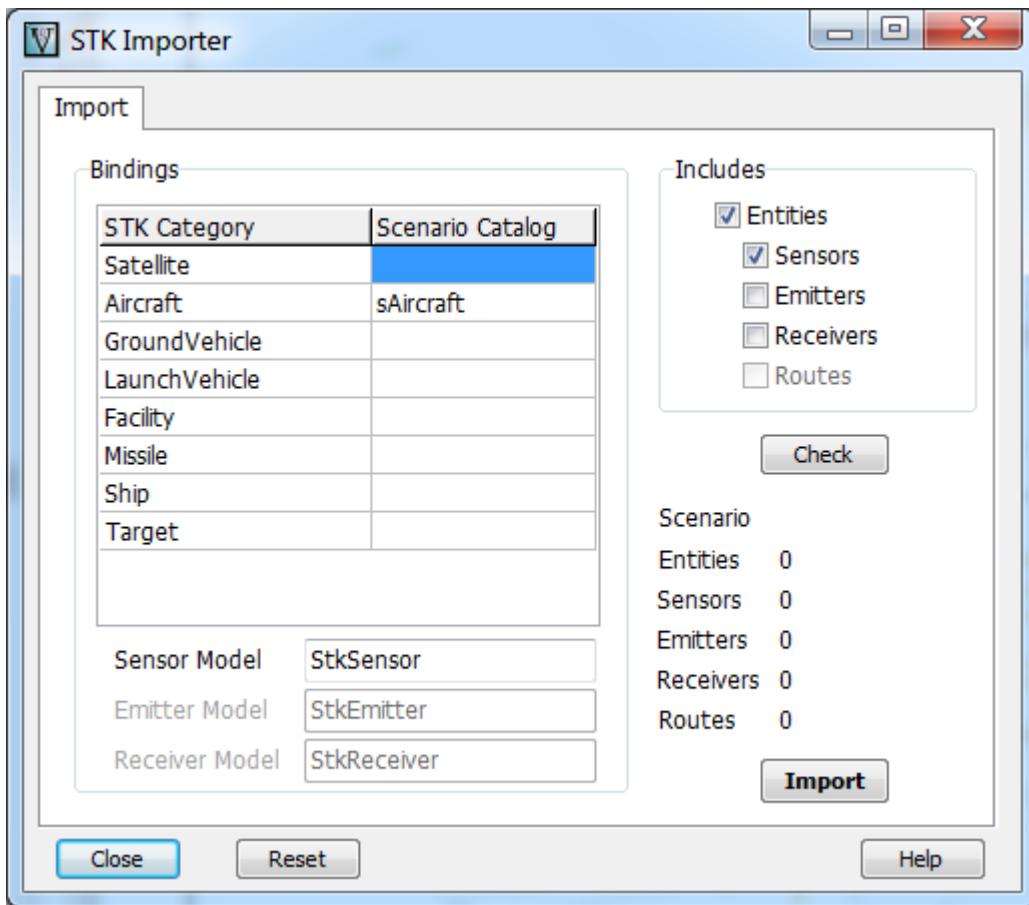
## Detections

# Detections

Add a sensor on STK aircraft



Update vsTasker scenario by importing the Sensor definition.  
On the [Includes](#) section of the STK Importer window, select [Sensors](#) for [Entities](#).  
Make sure that [Catalog](#) is correctly mapped with [Category](#).  
Then, [Import](#).



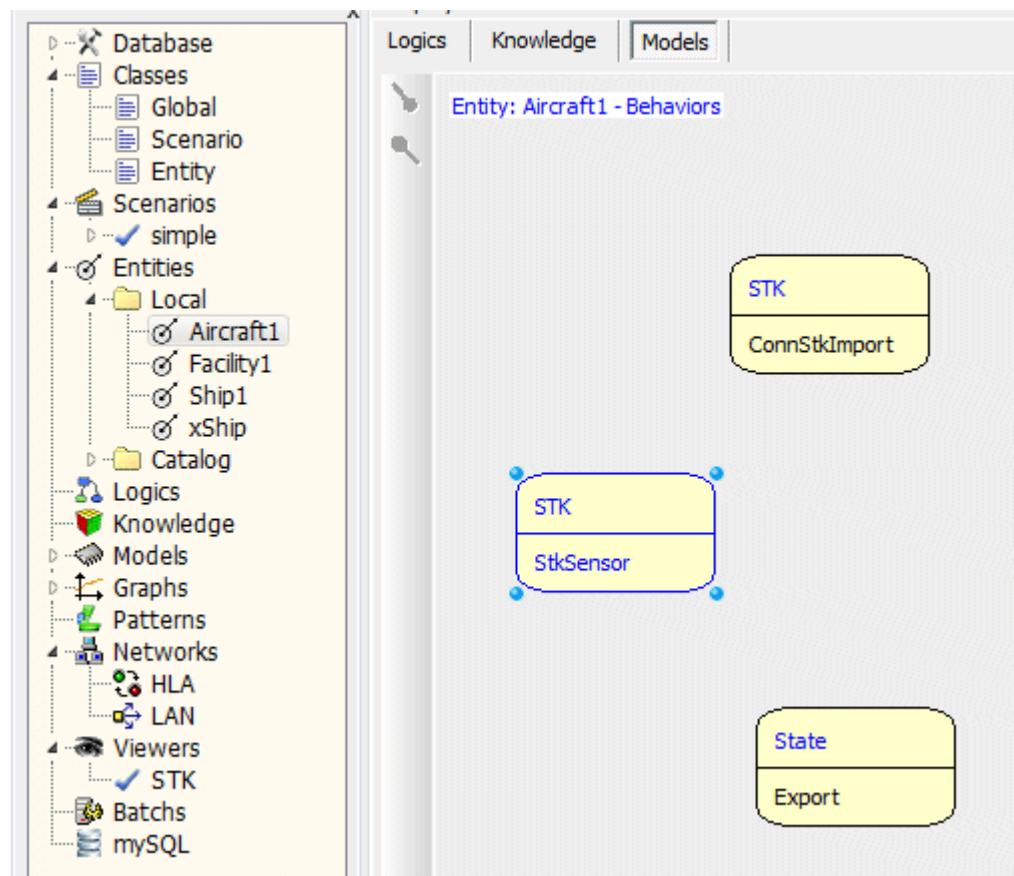
vsTASKER will add the [StkSensor](#) Model Component to every Entity having a Sensor defined in STK.

Name and some parameters will also be automatically set.

After Import, you can select any Entity with Sensors and check into the Models pane that [StkSensor](#) is loaded.

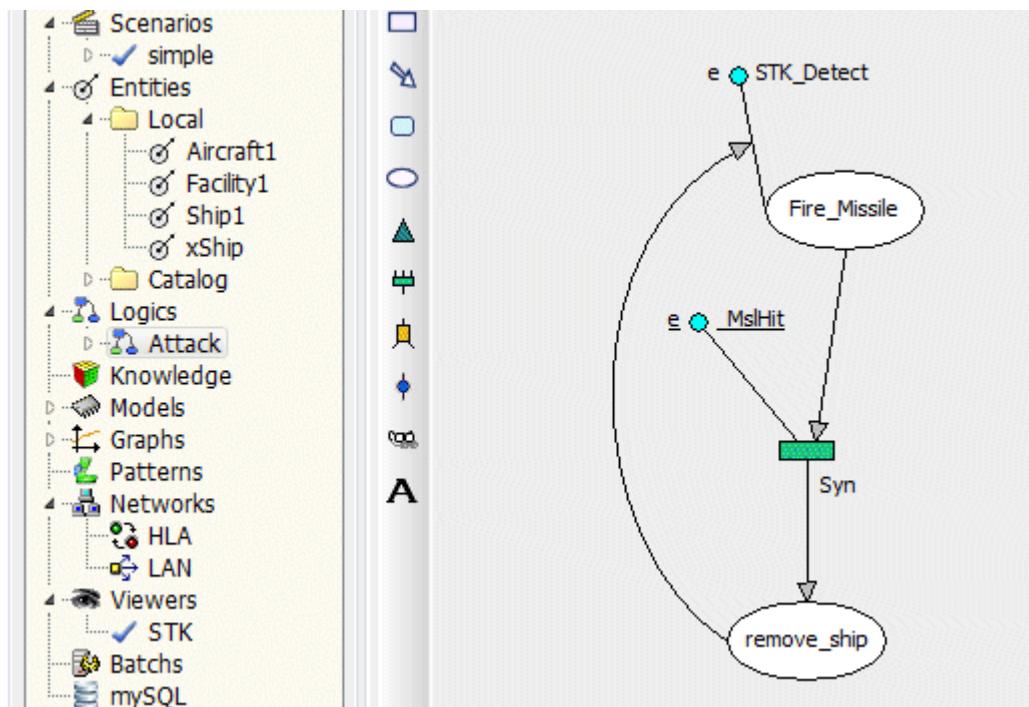
There is as many Component as Sensors.

## Detections



Now we can add a logic on vsTASKER that will react on anything detected by the STK Sensor.

Here, in case of event `STK_Detect` that is raised by the `StkSensor` whenever its counterpart in STK detects something, `Fire_Missile` Action will be triggered.



`Fire_Missile` Action will create a vsTASKER missile that will target whatever has been detected.

```
L:target = (Entity*) edata();

if (L:target) { // fire missile at
    StkMissile* missile = new StkMissile("xMissile");

    IAgMissilePtr miss=(IAgMissilePtr) missile->stkObject;
    IAgVOModelPtr model=(IAgVOModelPtr) miss->VO->Model;
    model->ModelType = AgEModelType::eModelFile;
    IAgVOModelFilePtr modelFile =(IAgVOModelFilePtr) model-
>ModelData;

    modelFile->Filename = strAdd(vsTaskerDir(), "\\\Runtime\\3D-
models\\harpoon\\harpoon.mdl");

    BasicMissile* missile_ctrl = missile->findBasicMissile();
    missile_ctrl->launch(ent(), L:target);
}
```

In case of destruction of the target (belonging to STK), both entities will be removed (or marked dead or wreck).

## Detections

```
sprintf(cmdString,"Graphics %s SetAttrType Basic",L:target->db->getProfile());
sendCommandToStk(cmdString);

sprintf(cmdString,"Graphics %s Basic Show Off", L:target->db->getProfile(), getEpochStk());
sendCommandToStk(cmdString);
```

Now, we can compile and run the scenario.

# MATLAB

- **Concept**

Matlab functions used in vsTASKER must be translated into C++ code and compiled with the MCC compiler of Matlab.

The compiler will produce a library that will gather all the Matlab functions defined in the database. It is this library that will be linked with the vsTASKER libraries to produce the simulation engine.

Matlab must be installed on the computer in order to run a simulation engine with Matlab functions. This is because the library produced by the MCC compiler relies on Matlab DLL that will be loaded with the vsTASKER simulation engine.

Once Matlab is installed, define the following environment variable that will point to the directory.

For i.e: `set MATLAB "D:\Program Files (x86)\MATLAB\R2011a"`

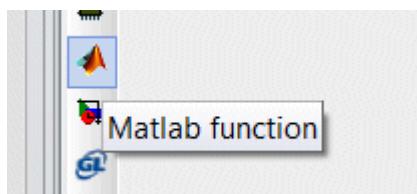
Open the **Tools::Preferences::MATLAB Settings** (see User Manual for description) and check the installation.

- **How to Use**

First, setup MATLAB environment (see above) to reflect the installation. Use Win32 MATLAB libraries if you expect to build a x86 simulation engine and Win64 for x64 bits platform engine.

Now, let's create a simple function `sqr` (for square root).

Create the Matlab function:



Then, write the function itself:

```
function y = sqr(a)
y = a*a
```

and in the Interface panel, add the following:

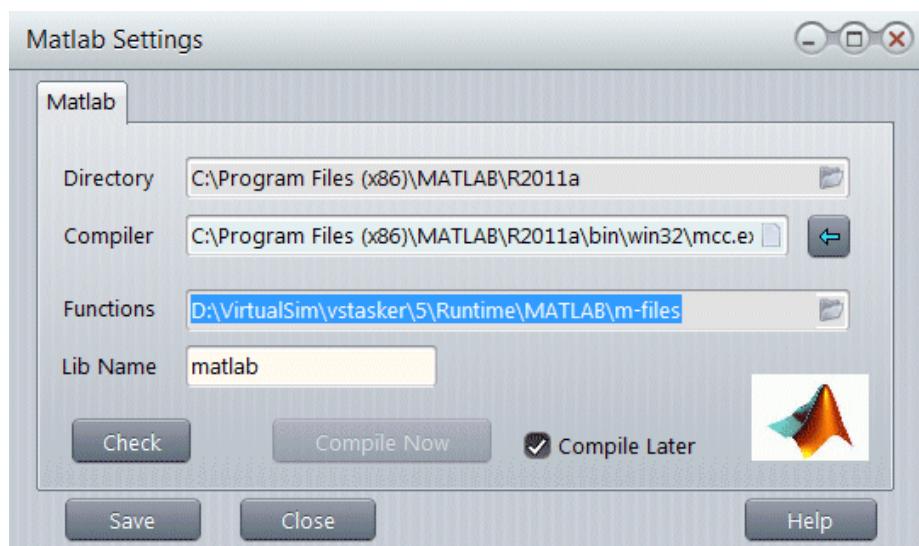
## MATLAB

```
return y scalar  
a scalar
```

Finally, with the  button, save the m-file as is:

D:\VirtualSim\vsTasker\7\Runtime\MATLAB\m-files\sqr.m

 *The directory (here \Runtime\MATLAB\m-files) must be the same as the one defined in the Matlab Settings for Functions (see below).*



Now, create a Logic [sqr\\_test](#) and one Action [sqr\\_display](#)  
In the Action, add the following code:

```
printf("Sqr(5) = %f\n", m.sqr(5)); // m is the accessor of all  
Matlab functions of the database
```

Give the [sqr\\_test](#) Logic to the (Scenario) Player then compile (you will see that the Matlab compiler will be called - if Matlab correctly setup), link and produce a Sim Engine.

Load the simulation and run.

On the Console, you will see: Sqr (5) = 25

# Simulink

- **Concept**

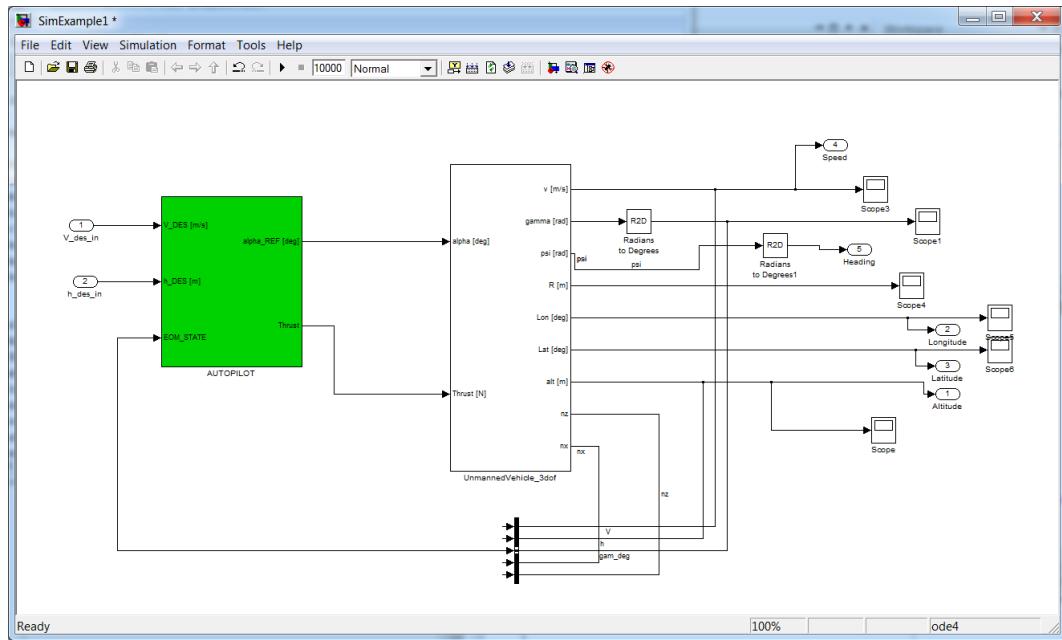
Simulink model must be generated using the [Simulink Coder](#) (use to be Real-Time Workshop) before being linked with vsTASKER libraries.

Once the code is generated, it will be compiled and linked.

Unlink with Matlab functions, Simulink model does not produce a library. All code will be compiled every time simulation engine will be generated.

- **How to Use**

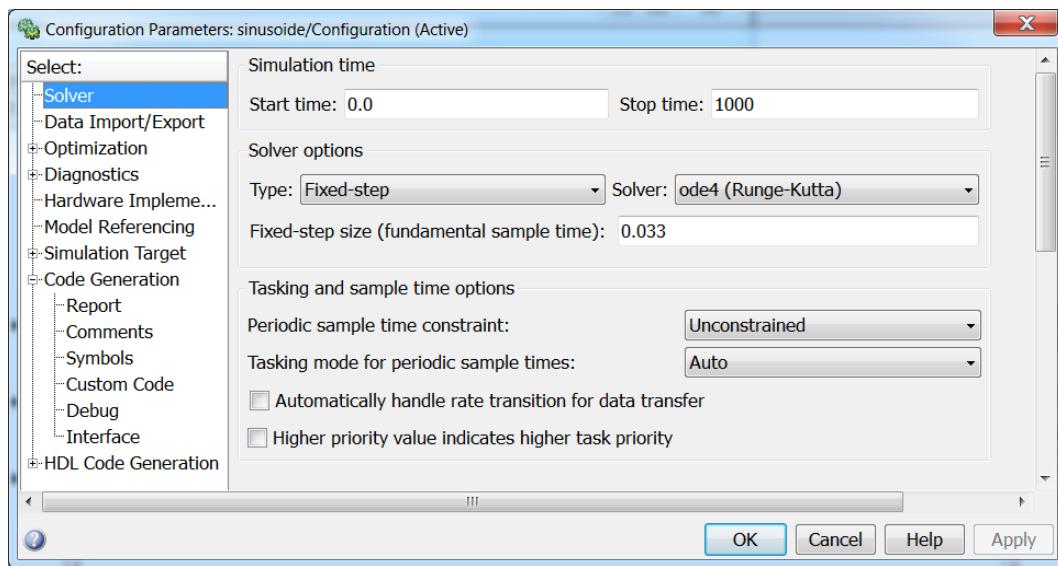
Open or create your model inside Simulink (here, [SimExample1.mdl](#))



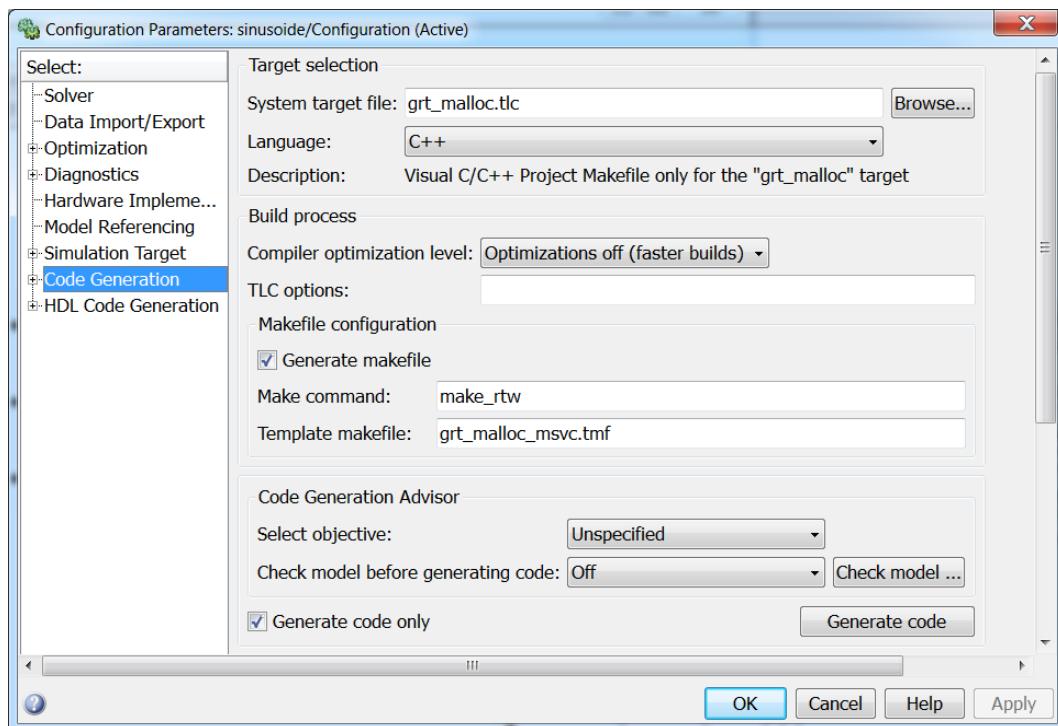
Setup the Code in order to produce a code that is compatible with vsTASKER libraries and use a proper solver.

[Tools::Code Generation::Options...](#)

## Simulink



In this window, set the [Solver](#) and the [Fixed-step](#) value (in seconds) that must be coherent with the setting in vsTASKER for the [Simulink object](#).



For the [Code Generation](#), use the above settings ([grt\\_malloc.tlc](#) for the [System](#), [C++](#) for the [Language](#) and [make\\_rtw](#) for the [Makefile](#))

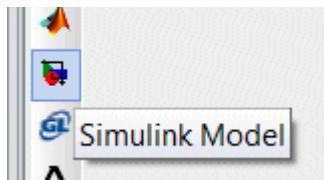
The generated code will be put on the working directory of Matlab (in Matlab, for example: `cd 'D:\VirtualSim\vsTasker\7\Runtime\Simulink\sinusoid'` to

set the working directory to point to the model `sinusoide.mdl` file; the generated code will then be there)



note: every time you do a change in the Simulink model, you will need to regenerate the C code and then, recompile the vsTASKER database to get a new simulation engine.

Create the Simulink object inside vsTASKER then set it.



## • Code Hints

**I**: is a macro that returns a pointer to the `Input` parameter structure of the Simulink model.  
**O**: is a macro that returns a pointer to the `Output` parameter structure of the Simulink model.

# GL-Studio

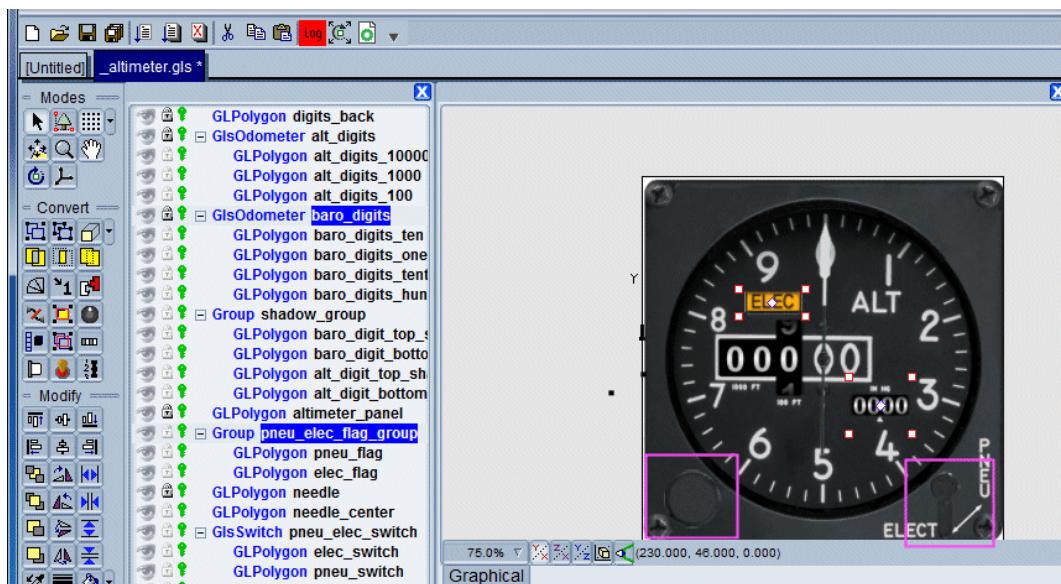
- **Concept**

GL-Studio outputs a C++ code that instantiate several graphic objects belonging to their own library. These graphic objects have either some input values, output values or both, depending of these categories.

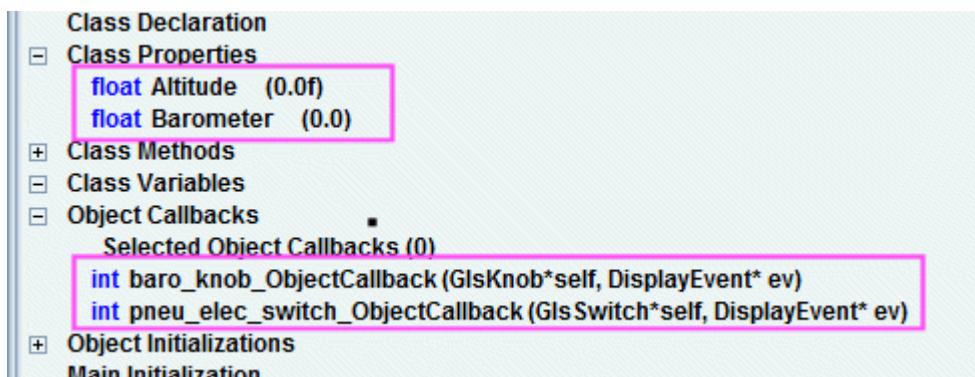
vsTASKER is able to mix his own generated code with the one generated by GL-Studio in order to actuate graphic objects. It also can be controlled from graphic objects.

- **How to Use**

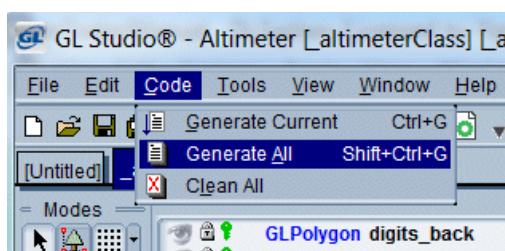
Open or create you model inside GL-Studio (here `_altimeter.gls`)



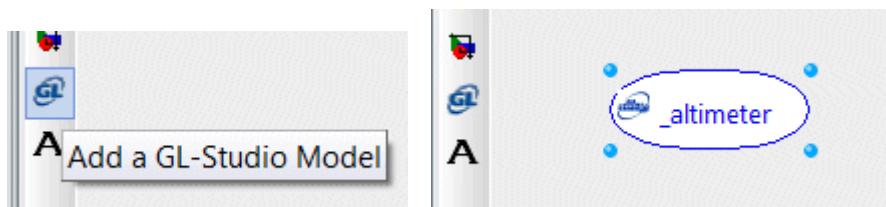
We will use the two variables to set the [Altitude](#) and the [Barometer](#), plus the two callback functions to react to user input on the ELECT/PNEU switch and the BARO knob (in magenta on the picture above)



GL-Studio can now generate the code:

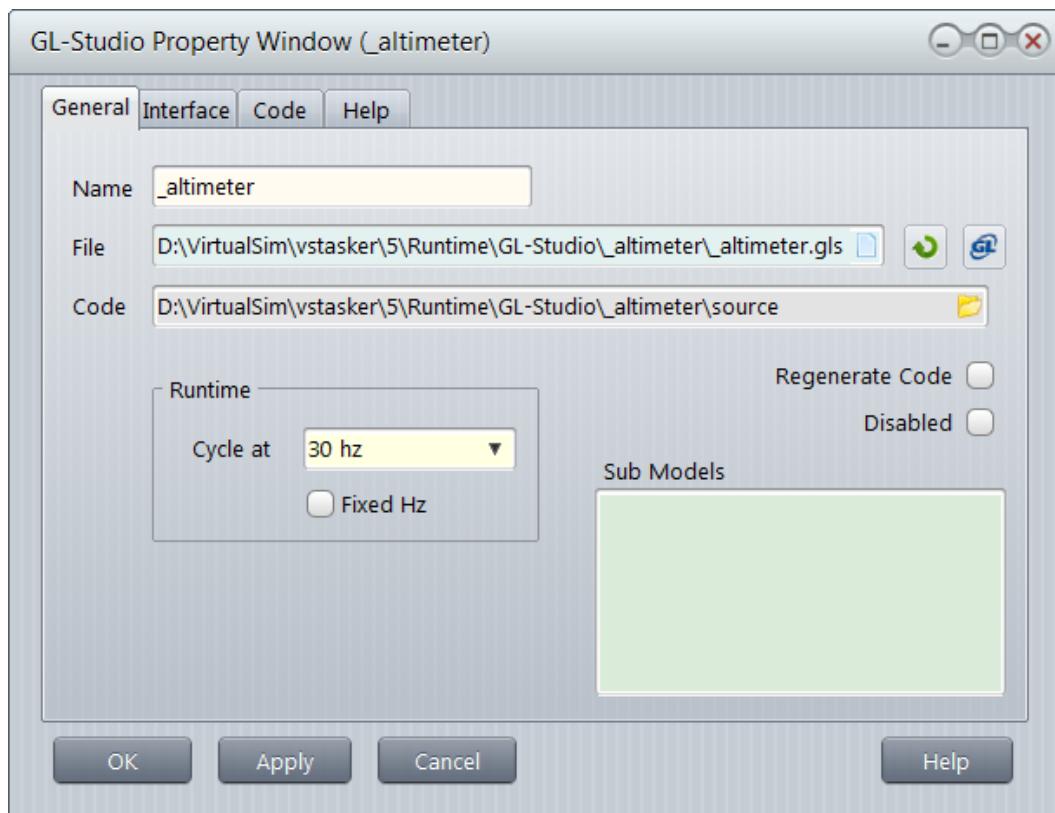


Create the GL-Studio object inside vsTASKER then open it:

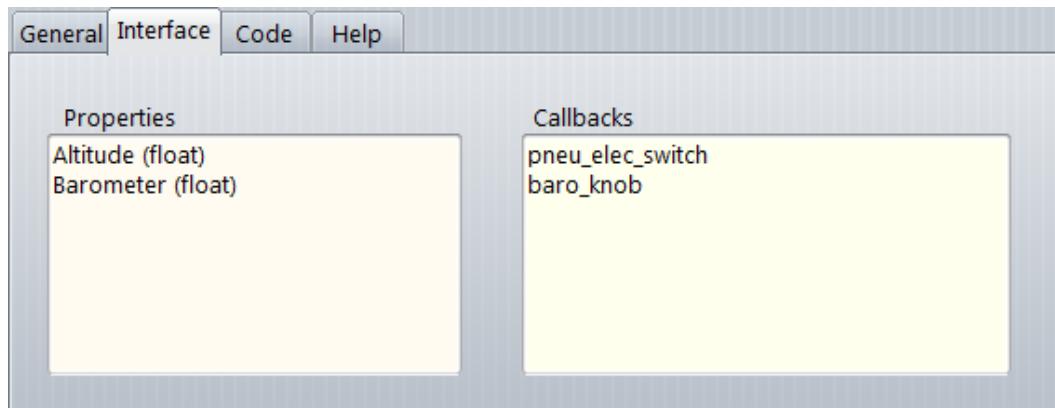


In the property window, set for **File** the GL-Studio model saved from inside GL-Studio application:

## GL-Studio



Then specify the source code directory (Code) where GL-Studio has put its generated code. This will be used to extract the interface:



Now, in the code, let's define two variables:

```
public:  
    float bar; // && DEF[28]  
    float alt; // && DEF[5000]
```

In the runtime part of the code, let's access the object (O:) variable and feed them with the actual entity values:

```
alt = E:getDyn()->getAltitude();  
  
O:Altitude(alt);  
O:Barometer(bar);
```

You now need to use a GL-Studio [viewer](#).  
See here on how to do.

Compile, generate then run.  
You will see the altimeter object output running nicely on its own little window.

## • **Code Hints**

**O:** is a macro that returns a pointer to the GL-Studio graphical object.

## **SQL**

# **SQL**

The SQL feature of vsTASKER allows easy access to a SQL server using C/C+ + API normally provided by the vendor. From the GUI, user can define Schemas, Tables and their content. Each table can be loaded at start or created and filled during the play. vsTASKER generates all necessary code to facilitate the access of the data and the connection to the SQL database.

Refer to the chapter below to learn how to setup the SQL environment according to the selected software used.

vsTASKER has been used with some SQL servers in order to import data or to store result for analysis or display. Only the following two are detailed.

# Using mySQL

First, you need to install the [mySQL Workbench](#).

[Go to http://dev.mysql.com/doc/workbench/en](http://dev.mysql.com/doc/workbench/en) and download [mySQL Workbench 5.7](#) and [Connector C](#)

Install the following modules:

- Workbench 8
- Server 5.7
- Connector.C 6.1

Make sure that the libraries and Windows SDK supported by Connector C matches the Visual Studio environment you are using.



**For the above environment, Visual Studio 2015 (vc140) and SDK 8.1 have been used.**

The x64 version of mySQL Connector C should be used with x64 libs of vsTASKER.

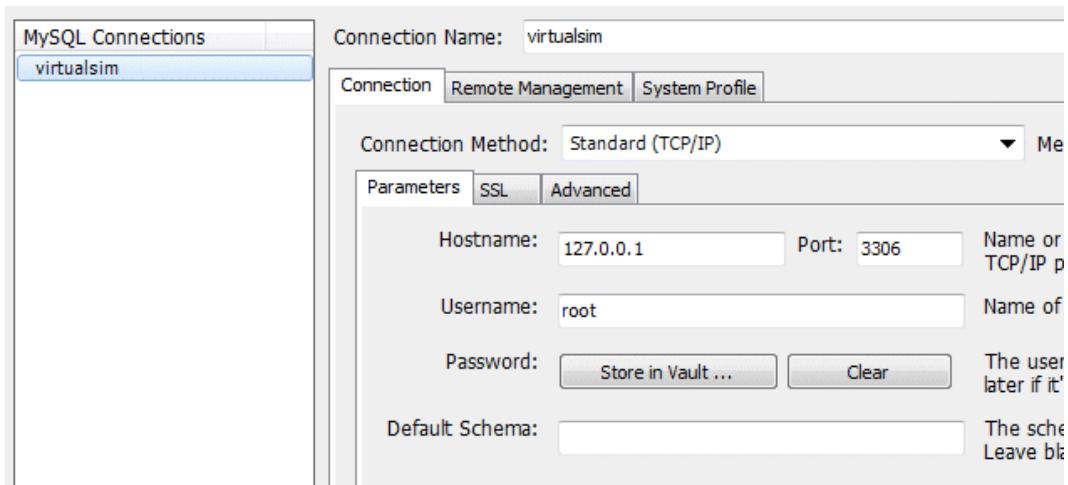
Set the environment variable `MYSQL_C_DIR` to [C:\Program Files \(x86\)\MySQL\MySQL Connector.C 6.1](#)

## • Setup

Start the **mySQL** server.

Start **Workbench**

Create a mySQL Connection:

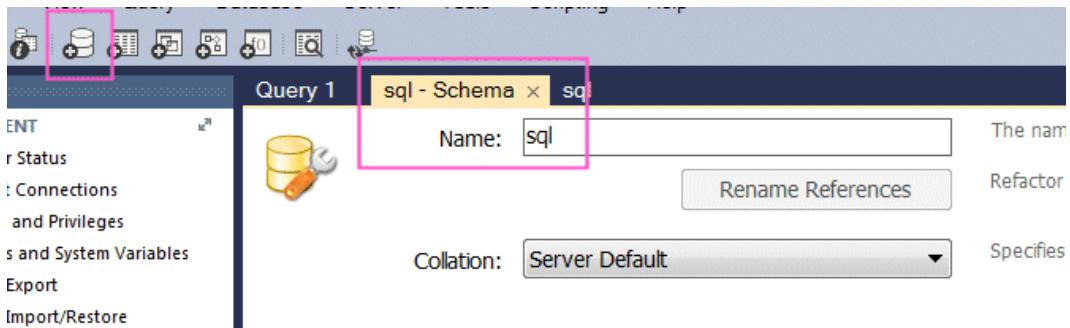


## Using mySQL

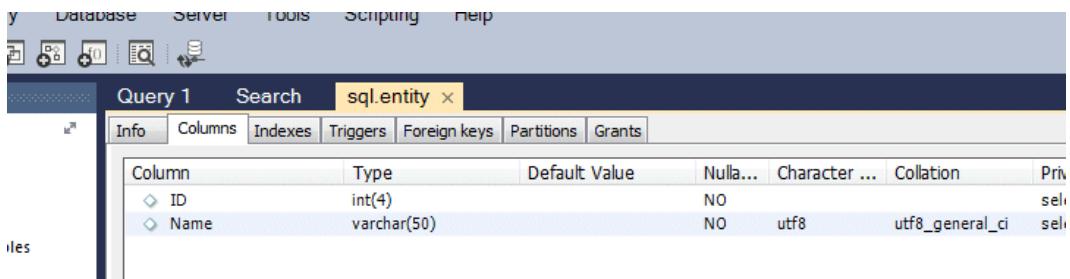


You can **Test Connection**. If failed to connect to mySQL, check in your services that **MySQL57** is up running (sometimes, in Manual mode, just Start it)

Create a schema and name it "**sql**":



Now, let's create a Table named "**entity**" with two columns (**ID**, auto-increment, and **Name**, character string)



Time to add some data into the table (3 entities).

We now add "**ent1**", "**ent2**" and "**ent3**" entity names into the table from the **Query** panel (and check the result below):

```
INSERT INTO sql.entity (NAME) VALUES ("ent1");
```

## Using mySQL

The screenshot shows the MySQL Workbench interface. In the top right, there's a query editor window titled "entity" containing three INSERT statements:

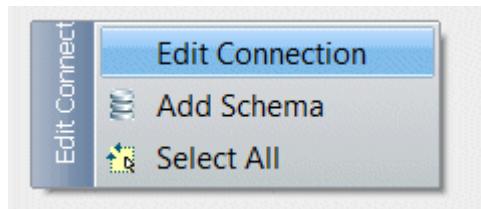
```
1 ✓ INSERT INTO sql.entity (NAME) VALUES ('ent1');
2 ✓ INSERT INTO sql.entity (NAME) VALUES ('ent2');
3 ✓ INSERT INTO sql.entity (NAME) VALUES ('ent2');
```

Below the query editor is a "Result Grid" showing the data inserted into the "entity" table:

ID	Name
1	ent1
2	ent2
3	ent3
*	HULL

The left sidebar contains navigation sections like "MANAGEMENT", "INSTANCE", "PERFORMANCE", and "SCHEMAS". Under "SCHEMAS", the "sql" schema is selected, and the "Tables" section shows "entity" and "result". A pink box highlights the "entity" table icon.

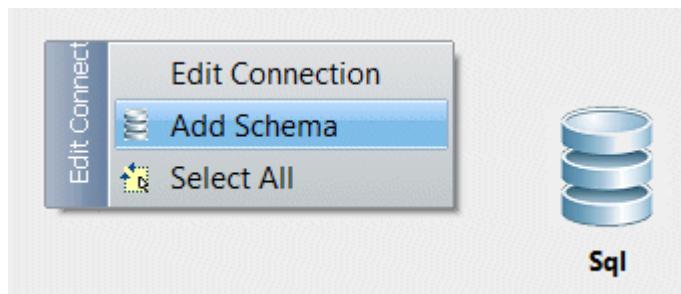
In vsTASKER, set the Connection with the same host, user and password set in Workbench for the setup



See here for the setting.

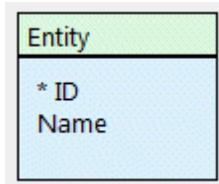
## • Reading from a Table

Now, in vsTASKER, let's add a schema and name it **sql**:



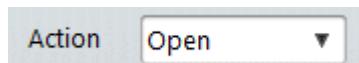
Create inside this schema a read-only Table for entity input.  
Let's call it "**Entity**":

## Using mySQL



This table will list all entities to add into the scenario and this table has already been created in Workbench below.

We need to set the **Action** to **Open**:

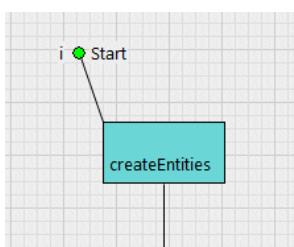


Then let's add 2 columns, **ID** and **Name**:

Two side-by-side screenshots of the MySQL Workbench Column Properties dialog. The left dialog is for column 'ID' with Type INT(0) and Length 4. The right dialog is for column 'Name' with Type VARCHAR(0) and Length 50. Both dialogs show checkboxes for Primary Key (checked), Not Null (checked), and Auto Incremental (checked). The 'Settings' tab is selected in both.

We need to create a logic that will query the table and automatically create the entities.

We will give this logic to the scenario player:



```
for (int i=0; i < db_sql.Entity.count(); i++)
{
    TSqlEntity* item = db_sql.Entity[i];
    WCoord pos;
    pos.setRandom(1000); // km around center
    Entity* ent = new Entity("basic", pos, item->Name);
```

```
    ent->activateNow();  
}
```

In the code above, we are using `db_sql.Entity` which is code generated. `db_sql` is a generated structure that will contain all the tables defined in the active schema.

Each Open table produces a class of the same name with all the columns translated into class members.

At simulation start, vsTASKER simulation engine connects to the database and read all the SQL tables (in [Open mode](#)) to fill the actual local instances.

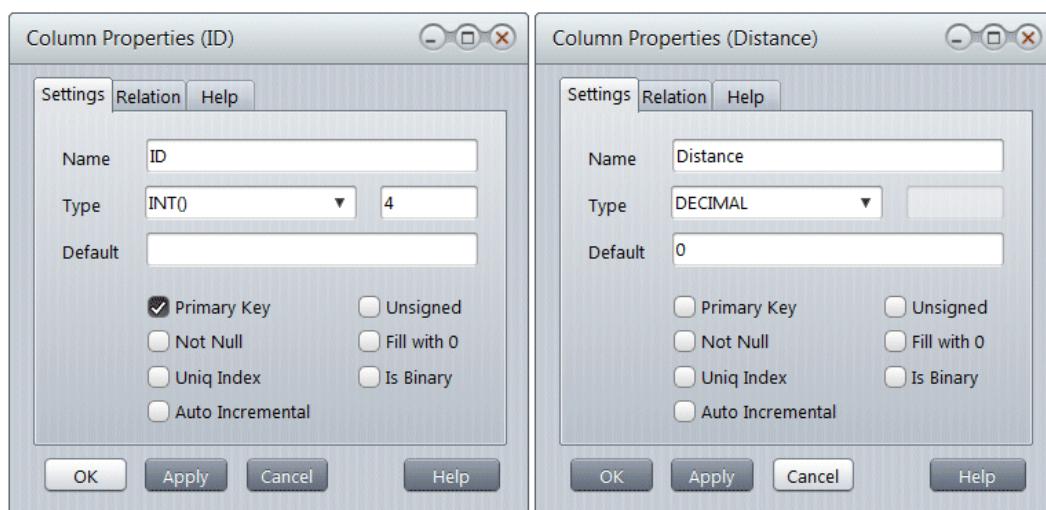
In the sample above, `db_sql.Entity` is an Array that will contain the three entries we added previously from Workbench (`ent1`, `ent2`, `ent3`) and they will be randomly positioned on the scenario by the Logic.

## • Writing into a Table

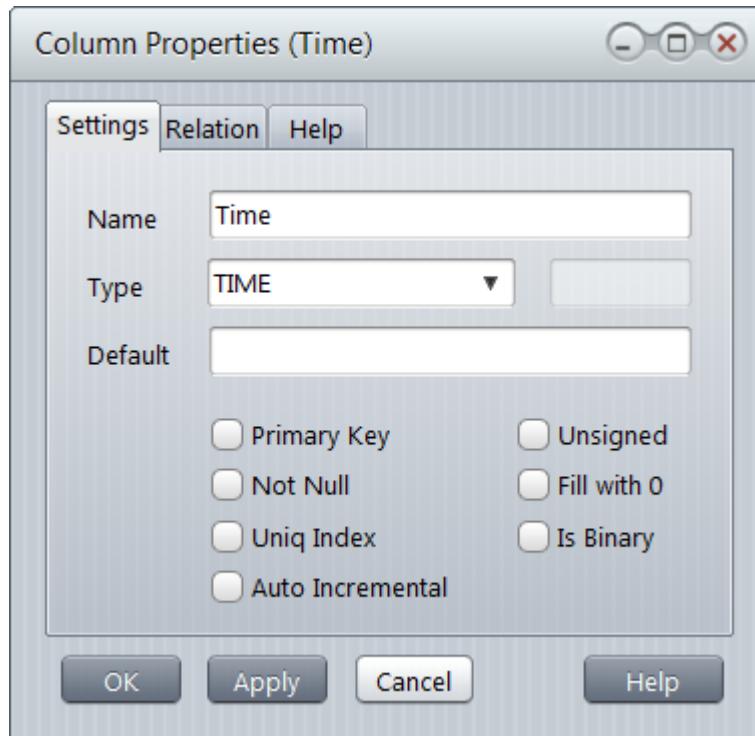
The same way as we did create the entity table in Workbench, as an input to vsTASKER, we will create one table inside vsTASKER for the mySQL database.

Let's call it "**Result**" and set the [Action](#) to [Create](#).

Now, we add three columns (**ID**, **Distance**, **Time**):



## Using mySQL



To insert result data into this table from vsTASKER source code, do the following, anywhere in a Logic or Component:

```
char msg[500];
sprintf(msg, "INSERT INTO Result (ID, Time, Distance)
VALUES ('%d', '%d', '%f')",
        E:getId(), E:chrono.read("time_to_point"), L:dist);
mysql_query(db_sql.ptr, msg);
```

use `mysql_query` with the pointer to the SQL database and the message where you will put your SQL command using the standard syntax.



*When running vsTASKER simulation engine connected to a MySQL database, if the simulation just die at launch, check the file /sim\_out.txt for explanations. Often, the connection to the database could not be done (server not running, schema not found, permissions failed...)*

# mySQL Sample

This demo works with **mySQL** (Workbench 5.2).

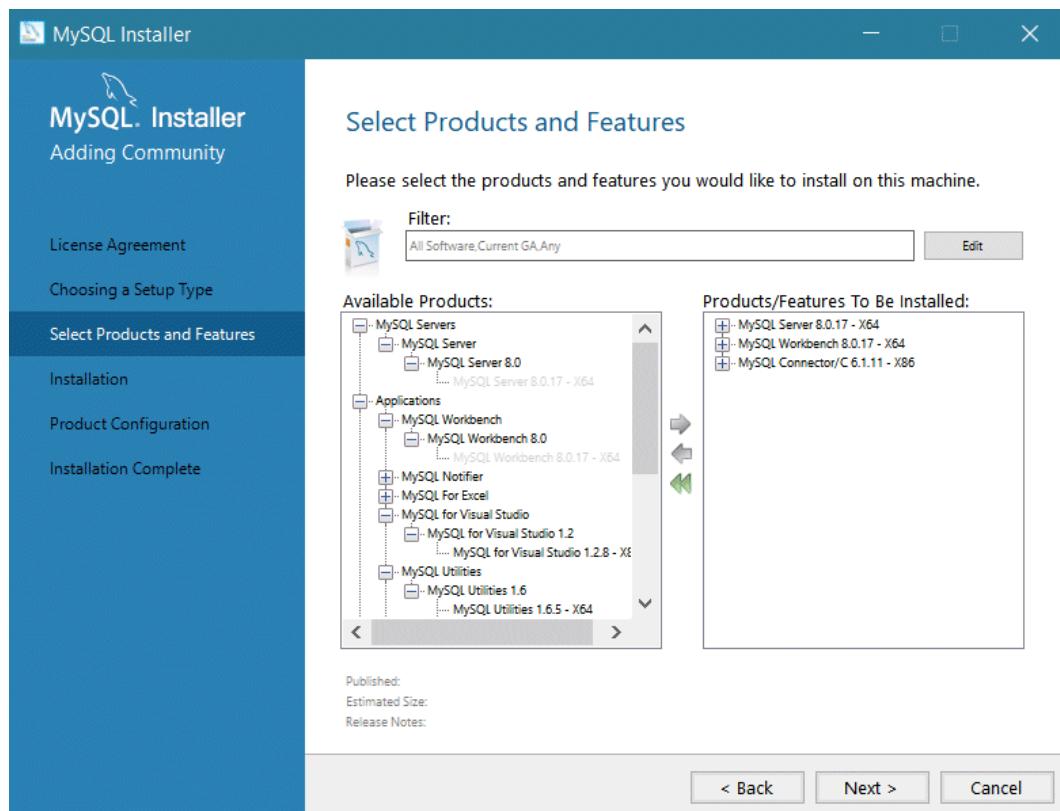


## • Prerequisite

Install the following software:

- MySQL Community Server
- Connector/C (libmysql)
- MySQL Workbench (GUI Tool)

<http://dev.mysql.com/downloads/mysql>



**NOTE** You must set **MYSQL\_DIR** to point to "C:\Program Files\MySQL\MySQL Server <version>"

You must set **MYSQL\_C\_DIR** to point to "Program files (x86)/mySQL/MySQL Connector C <version>"

## Simple Test

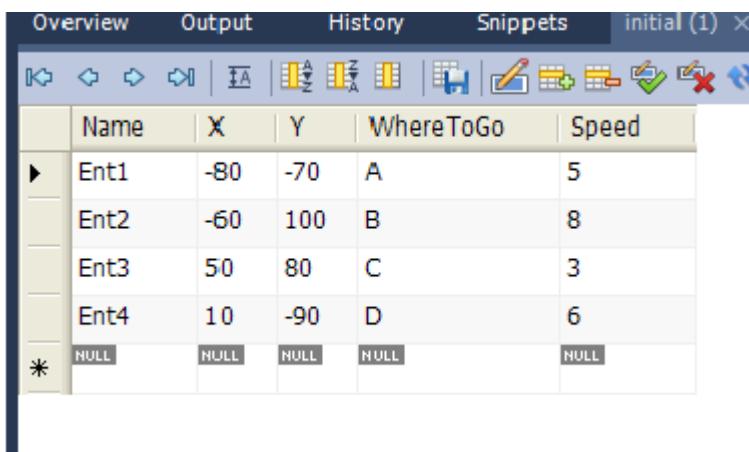
# Simple Test

Open the scenario `SQL/test_mysql` or `SQL/test_sqlite` as the two databases are similar but are running with different SQL servers and API.

All tables are located in the Schema SQL.

Open MySQL Workbench, connect to the database and import the `test_mysql.sql` auto-extract dump from `/Runtime/SQL/mysql` directory

You can check in Workbench that the entity table contains the following data.



	Name	X	Y	WhereToGo	Speed
▶	Ent1	-80	-70	A	5
	Ent2	-60	100	B	8
	Ent3	50	80	C	3
	Ent4	10	-90	D	6
*	NULL	NULL	NULL	NULL	NULL

Recompile the simulation engine and start it. You will see the four entities being created at the specified coordinates (from the table) and start moving towards their named locations, with the given speed.

## • How it works

The Open mode SQL Tables are automatically loaded by the simulation engine and the content is stored into a code generated data structure, matching the row definitions of the Table.

In this example, the initial table has generated a class `Initial` accessible using the structure named `db_sql`. The `Initial` class has all column values as members, with the same name. You can see how the Player extracts the entities from the table `Initial` and create them. Logic: `CreateEntities`

Later, each entity will write specific data into the table `Result` using a simple SQL command: Action `StoreDb` in `WhereToGo` Logic.

At the end of the simulation, the result table can be parsed (from the Workbench) to see the values written by the simulation engine during the play.

## **Simple Test**

# Using MariaDB

MariaDB is a copy-like version of mySQL, so it is almost plug and play for systems which are mySQL ready.

Nevertheless, the download of the server and the C++ connector must be done from another source.

First you need to install the [MariaDB SQL server](#).

Go to [mariadb.com](#) and install the Community (free) server.

Then select [Connectors](#) tab (in the MariaDB website) and download the [Connector C](#) (version 32 or 64 bits according to the library you intend to use with vsTASKER. You can download both if you plan to use both version. vsTASKER will select the correct one according to the compiler setting)

Now install the SQL server and the C connector.

Follow the MariaDB installation procedure.

Now, set the following environment variable: `MARIADB_C_DIR` to `C:\Program Files (x86)\MariaDB\MariaDB Connector C`



The x64 version of MariaDB Connector C can also be used to compile with x64 libs of vsTASKER. In such case, the environment variable should be set accordingly:

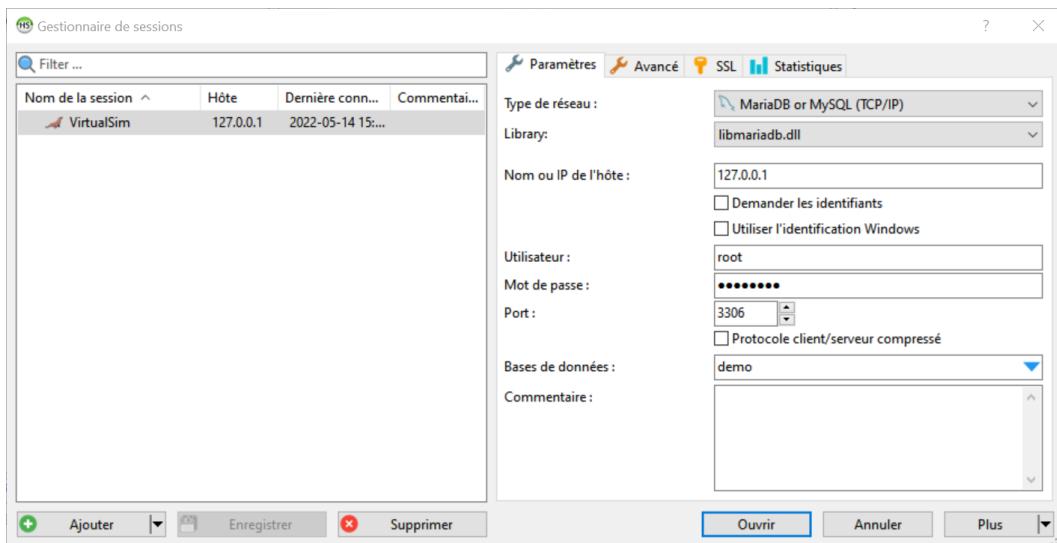
`MARIADB_C_DIR` to `C:\Program Files\MariaDB\MariaDB Connector C 64-bits`

## • **setup**

To test the `test_mariadb` sample, you need to create a session.

Open [HeidiSQL](#).

Name the session whatever you want, and give the root password you setup on the server side. Select the database demo:



Then, open the session (and the database demo) and add the following request to create the Initial table:

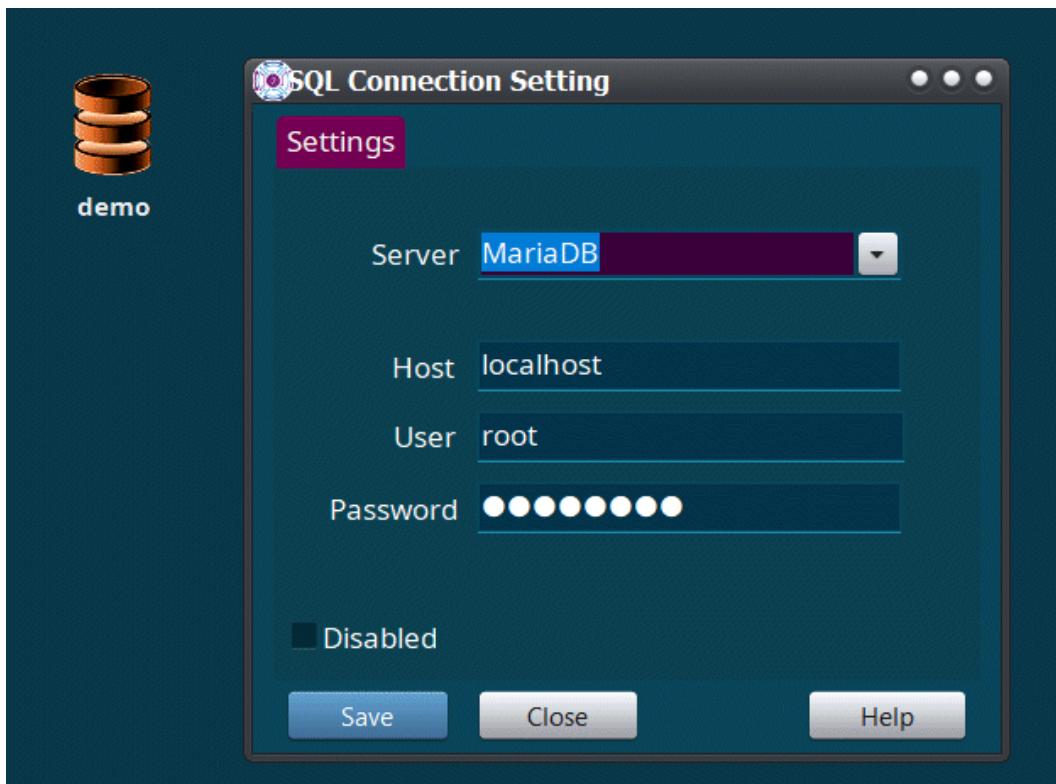
```
DROP TABLE IF EXISTS `initial`;
CREATE TABLE `initial` (
  `Name` varchar(45) NOT NULL,
  `Latitude` decimal(10,0) DEFAULT NULL,
  `Longitude` decimal(10,0) DEFAULT NULL,
  `WhereToGo` varchar(5) DEFAULT NULL,
  `Speed` decimal(10,0) DEFAULT NULL
);
```

Then, with the following request, fill the table with entity position entries:

```
INSERT INTO `initial` VALUES ('ent1',-80,-70,'A',5),
('ent2',-60,100,'B',8), ('ent3',50,80,'C',3), ('ent4',10,-90,'D',6);
```

Now, start vsTASKER, open the **SQL/test\_mariadb** database, and setup the SQL server:

## Using MariaDB



the generated code will create a session on the [localhost](#), under user [root](#) and [password](#).

The associated database (name) is here [demo](#) (Schema name)

Compile the database and start the simulation.

You should see four entities starting at various positions on the map. These positions have been extracted from the [Initial](#) table.

vsTASKER will create two tables: [entity](#) and [result](#).

You can see them on *HeidiSQL* client once the simulation ends (you need to refresh or start a new session to see these tables).



## Using MariaDB

Now, you can have a look at the tables, how they are defined, and the logics, for how to read data from and save data to tables.

## Using SQLite

# Using SQLite

SQLite is a light SQL server which is good enough for using standard SQL database. Nevertheless, the syntax for the SQL commands from the code (character strings) is more sensitive than with mySQL. Also, as it does not include a server, only one client can access a database file.

First, you need to install the SQLite Browser.

[Go to https://sqlitebrowser.org/](https://sqlitebrowser.org/) and download the latest release. Install it.



vsTASKER provides the [SQLite3](#) C/C++ libraries for supported Visual Studio version. Include `/SQL/sqlite3.h` and link against `sqlite3{d}.lib`

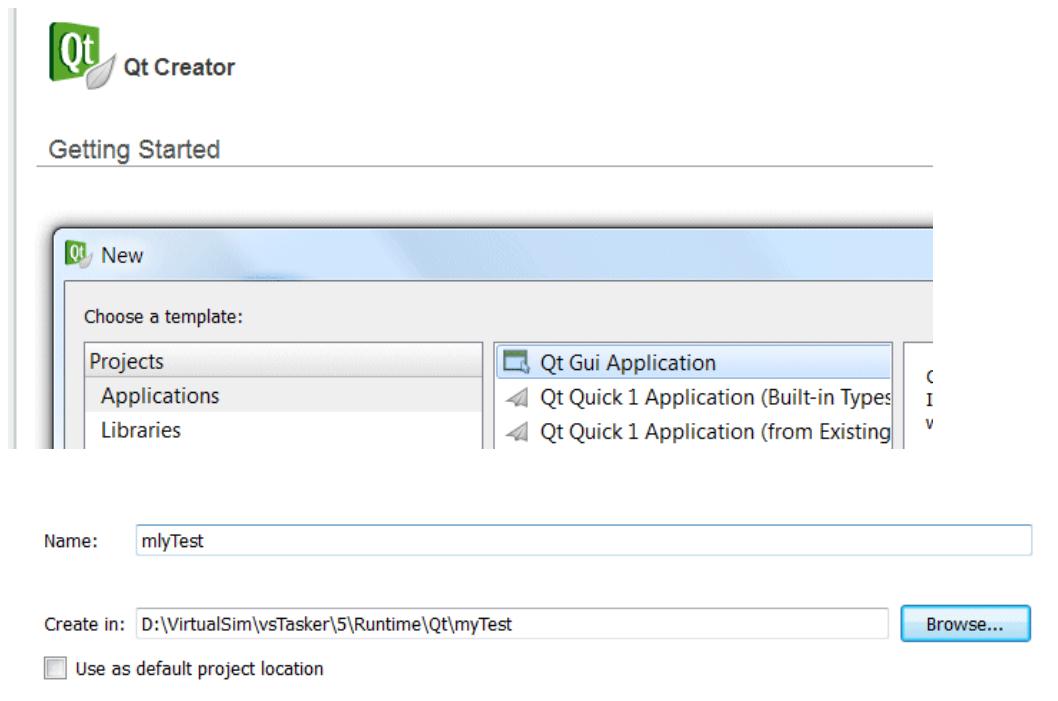
- **Setup**

# Qt

- **How to Use**

In this sample, we will create a simple **Qt Window** with one button to activate a logic with one lamp (that turns green when a logic starts).

In Qt Creator, create a [Qt Gui Application](#):



Click **Next** until the end of the setup.

Now, in **main.cpp** file, add the following:

```
#include "vt_rtc.h"
Vt_RTC* vt_rtc = NULL;

// =====
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    ...
}
```

in **mainwindow.cpp** file, add the following:

## Qt

```
#include "QDebug"
#include "vt_rtc.h"
#include "myTestQt.h" // vsTasker database header
```

then change the **MainWindow** constructor with the following one:

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    vt_rtc = new Vt_RTC;
    autorun();

    connect(&loop_timer, SIGNAL(timeout()), this,
    SLOT(loopTimer()));

    loop_timer.setInterval(33); // 33 milliseconds, 30hz
    loop_timer.start();

    ui->setupUi(this);
}
```

and the destructor with this one:

```
MainWindow::~MainWindow()
{
    delete ui;
    delete vt_rtc;
}
```

Now, let's define the vsTASKER runtime loop (called above **loopTimer**):

```
void MainWindow::loopTimer()
{
    static int lb=0, s=0;

    if (++lb==30) { // low band 1hz, always useful for tracing
        lb = 0;
        qDebug() << ++s;
    }

    if (vt_rtc->isOn()) {

        vt_rtc->tic(); // mandatory, to call the runtime node cycle
    }
}
```

```

    if (vt_rtc->isRunning()) {
        ui->startLogic->setEnabled(true); // allow button to be
used
        // get the entity
        Vt_Entity* myEnt = vt_rtc->scenario->findEntity("myEnt");
        if (myEnt) { // find the logic
            Vt_Logic* start = myEnt->findLogic("myLogic");
            if (start->isEnabled()) ui->logic-
>setStyleSheet("background-color: rgb(0, 255, 0)"); // green
            else ui->logic->setStyleSheet("background-color:
rgb(100, 100, 100)"); // gray
        }
        else ui->startLogic->setEnabled(false); // must start the
simulation first
    }
    else {
        exit(0); // we have a problem !
    }
}

```

In **mainwindow.h**, add the following code emphasized in bold:

```

#include <QTimer>

private slots:
    void loopTimer();
private:
    Ui::MainWindow *ui;
    QTimer loop_timer;

```

Now, in vsTASKER, create a new database, called **myTestQt** and save it

Back in Qt Creator, right click on the **myTest** project and **Add Existing Items...**  
Select in **/gen/** the three files: **myTestQt.h**, **myTestQt\_code.cpp** and  
**myTestQt\_intf.cpp**

Now, open the **myTest.pro** and add the following:

```

QMAKE_CXXFLAGS += -Zc:strictStrings-

INCLUDEPATH += . \
    $(VSTASKER_DIR)/include \
    $(VSTASKER_DIR)/include/engine \
    $(VSTASKER_DIR)/feature/include \

```

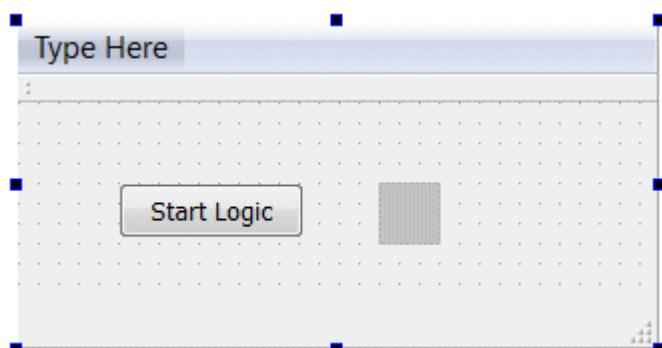
## Qt

```
$ (VSTASKER_DIR)/sprite/include \
$ (VSTASKER_DIR)/plugins/include \
$ (VSTASKER_DIR)/model/include \
$ (VSTASKER_DIR)/src/template \
$ (VSTASKER_DIR)/gen

LIBS += -L$(VSTASKER_DIR)/Lib/vc90/ -lglut32 \
        -L$(VSTASKER_DIR)/Lib/vc90/ -lsdl \
        -L$(VSTASKER_DIR)/Lib/vc90/ -lsdl_image \
        -L$(VSTASKER_DIR)/Lib/vc90/ -lvstasker \
        -L$(VSTASKER_DIR)/Lib/vc90/ -lvstasker_sim \
        -L$(VSTASKER_DIR)/Lib/vc90/ -lspattern \
        -L$(VSTASKER_DIR)/Lib/vc90/ -lspzone \
        -L$(VSTASKER_DIR)/Lib/vc90/ -ltraj \
        -L$(VSTASKER_DIR)/Lib/vc90/ -lpath \
        -L$(VSTASKER_DIR)/Lib/vc90/ -lpoint \
        -lwsock32 \
        -luser32 \
        -lgdi32 \
        -lopengl32 \
        -lglu32 \
        -lfreetype \
        -lorbitcore \
        -lorbittools
```

Now, open the [mainwindow.ui](#) and add one button, called `startLogic` and one QWidget called logic and give it the following style-sheet: `background-color: rgb(100, 100, 100);`

You should then have:

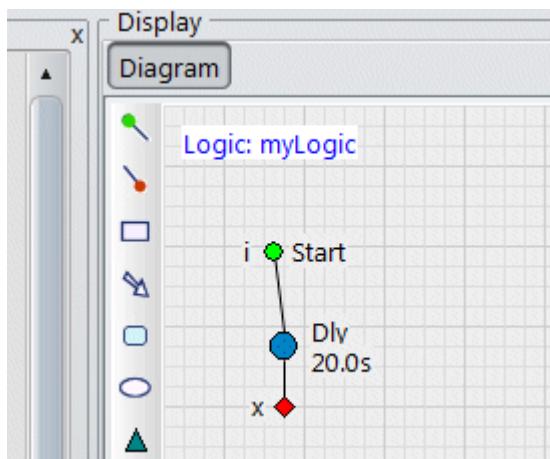


Now, right click on the `startLogic` widget, **Go to slot... `clicked()`**, then add the following code:

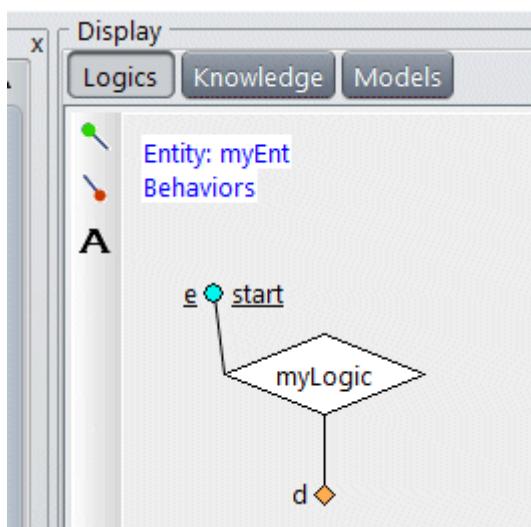
```
void MainWindow::on_startLogic_clicked()
{
    if (vt_rtc->isRunning()) {
        // get car entity
        Entity* myEnt = (Entity*) vt_rtc->scenario-
>findEntity("myEnt");
        if (myEnt) myEnt->raiseEvent("start");
    }
}
```

Now, in vsTASKER, add one entity and name it **myEnt**

Create one logic "myLogic" with the following (one delay set with a fix time of 20 seconds and a **Perform Quit** action.



Give this logic to **myEnt** with an activation on event "**start**". **Save** but do not compile yet.

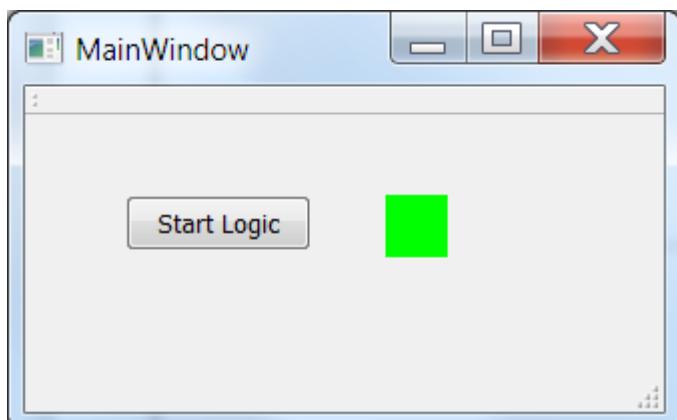
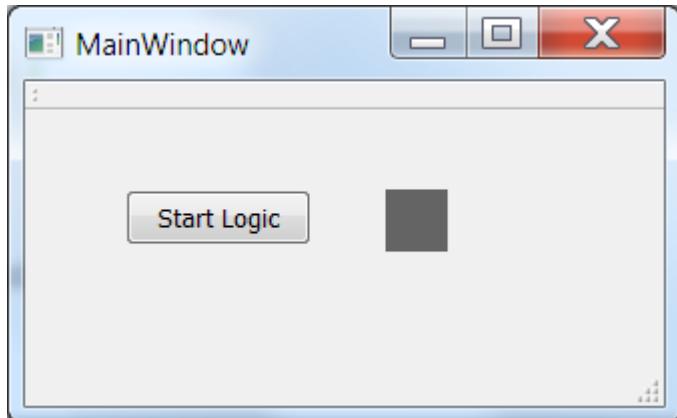


Back in Qt Creator, build and run in debug mode (good practice):

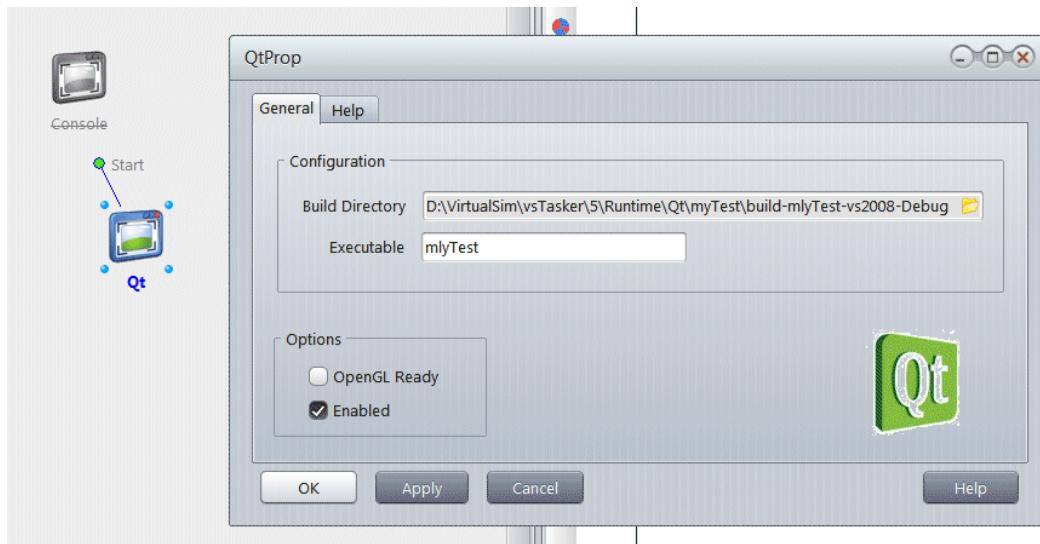
## Qt

In vsTASKER GUI, start running the scenario.

On the UI window (below), click the **Start Logic** button and see the lamp changing from gray to green then after 20 seconds, back to gray (when the logic ends)



When things work, you can add the Qt Viewer on the vsTASKER database.  
Add it normally, then set the [Build Directory](#) where Qt Creator stores generated code  
of your [myTest](#) project:



You can now start adding new logics and rebuild the window application like you would do with another viewer.

Use [Qt Creator](#) anytime you need to change the UI or change the way it must react with the simulation engine.

## Simple Remote

# Simple Remote

The simulation engine generated by vsTASKER can be controlled from another application using the shared-memory.

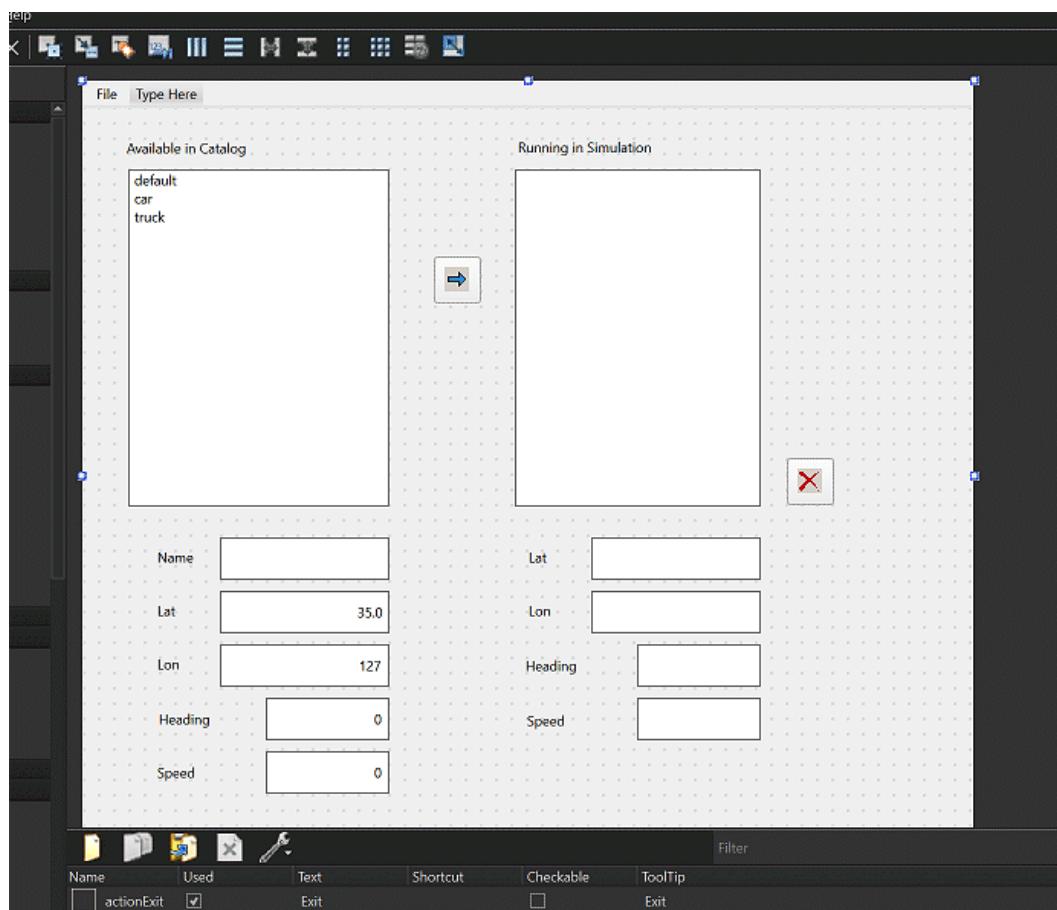
Refer to the detailed document in the Developer Guide.

In this sample, the database must match the UI counterpart, on the Catalog side (although this could be automatically done if the UI is able to load the database itself, but this would be more an integration than a connection).

Start vsTASKER and open the database Data/Db/Qt/simple\_remote/simple\_remote.db

The database seems empty but it contains 3 entities in Catalog (default, car and truck) and one Logic (load) which is attached on the car and truck and displays on the console basic information about the entity (name, position, heading, speed).

Start Qt and load the project in /Runtime/Qt/simple\_remote/simple\_remote.pro



In Qt, a very simple UI has been design for the purpose of this demo.

On the left, a Catalog list (prefilled) and below, fields to setup the initial values of an entity before instantiation.

On the right, a Runtime list with all instantiated entities and below, the actual values for any selected entity of the list.

➡ to instantiate a new entity if one in Catalog selected and a Name given. If none selected, the button will update the named entity (must be instantiated first).

✗ to delete the selected entity.

## • Libraries

To simply connect to the shared memory segment of vsTASKER, one library is enough: vcc\_gui.lib. Use the correct one according to your compiler (x86 or x64, vc10 or vc14).

The library vcc\_util.lib provides some useful libraries, although it is not mandatory.

### Connection

These includes are mandatory for the connection. Runtime.h (refer to the Developer Manual for a detailed description) is also requested by some types and API.

```
#include "runtime.h"
#include "shdmem.h"
#include "shmcomm.h"
```

These pointers give access to some specific segments of the shared memory.

The classes are defined in shmcomm.h.

```
RtcSharedMem* rtc_shm = NULL;
UpdSharedMem* upd_shm = NULL;
ShmKeyList    shm_keys;      // all shared-memory keys used. Def in
                           shdmem.h

RtEvents      gui_events;   // gui  -> sim
RtEvents      sim_events;   // sim  -> gui
RtEntities    rtent_pool;   // sim <-> gui
RtSharedMem   shared_memory;
```

Now, let's connect these pointers to the shared memory. If not created, the `initSharedMem` call will create, otherwise, it will simply connect to.

```
// get the access key
int shmkey;
if (getenv("VST_RTC_SHMKEY")) shmkey =
strtol(getenv("VST_RTC_SHMKEY"), 0, 16);
else shmkey = VST_RTC_SHMKEY;
```

## Simple Remote

```
// Create/Connect the shared memory
rtc_shm = (RtcSharedMem*) shared_memory.initSharedMem(shmkey,
sizeof(RtcSharedMem));
shm_keys.add(shmkey, "rtc", (long)rtc_shm);

if (!rtc_shm->sim_mode) rtc_shm->sim_mode = RTC_Idle;
rtc_shm->gui = ON;

// get the access key
if (getenv("VST_UPD_SHMKEY")) shmkey =
strtol(getenv("VST_UPD_SHMKEY"), 0, 16);
else shmkey = VST_UPD_SHMKEY;

// Create/Connect the shared memory
upd_shm = (UpdSharedMem*) shared_memory.initSharedMem(shmkey,
sizeof(UpdSharedMem));
shm_keys.add(shmkey, "upd", (long)upd_shm);

upd_shm->entities.on = true;

// Runtime Moving Entities dynamic shared-memory segment
if (!rtent_pool.initialized()) rtent_pool.init(upd_shm, 100); // 100 slots

// SIM -> GUI Runtime Event dynamic shared-memory segment
if (!sim_events.initialized()) sim_events.init(upd_shm, "sim", 50); // 50 slots

// GUI -> SIM Runtime Event dynamic shared-memory segment
if (!gui_events.initialized()) gui_events.init(upd_shm, "gui", 10); // 10 slots
```

## • Sending Commands

The UI will send commands by using a unique function with a specific data structure defined in runtime.h

The list of all commands is available in the Developer Manual.

Let's see here how to create an entity using a remote command, and how to set speed and heading.

How to create an entity from a remote command.

## Simple Remote

```
EvtData edata; // the main structure. ill use the new_ent union part  
WCoord npos(WC_LLA, lat, lon, 0); // position of the entity on the gaming area  
npos.convertToXYZ(); // must be converted into XYZ  
strcpy(edata.new_ent.base, "Entity"); // class name (see Entity classes)  
strcpy(edata.new_ent.catg, "car"); // catalog name  
strcpy(edata.new_ent.name, "my car"); // name of the instance  
edata.new_ent.x = npos.x; // position x  
edata.new_ent.y = npos.y; // position y  
edata.new_ent.z = 0; // altitude  
// Send the event to the sim  
gui_events.push("_NewEnt", "", edata);
```

How to change the speed of a runtime entity

How to change the altitude of a runtime entity

## • Receiving Events

The simulation engine can also send messages to the remote UI. These messages can result of UI commands but also from the normal course of the simulation. The UI must react to the most important of them in order to remain in sync with the engine.

New entity creation

Entity deletion

## • Retrieving entity data

## • Going further

## Qt + HMI

In vsTASKER, adding an HMI generates automatically code based on Glut and OpenGL. It is not necessary to add much code to see the HMI panel being displayed as a separate window with its own thread.

In the loop timer, add after the `vt_rtc->tic()`, add the following call:

```
vt_rtc->ticHmi();
```

That's it. You now have the HMI window displayed and usable.

## Qt + OSG

In vsTASKER, add an OSG Viewer and enable it. If the Qt viewer is also enabled, it will connect with a dashed line. This means OSG viewer object will be setup. Do not forget to setup the 3D terrain and to assign to each entity their 3D models.

Then, on Qt, add the following to the **.pro** file:

```
DEFINES += _VOSG

LIBS += -L$(VSTASKER_DIR)/Lib/vc140/ -lvcc osg \
        -L$(VSTASKER_DIR)/Lib/vc140/Osg -losg \
        -L$(VSTASKER_DIR)/Lib/vc140/Osg -lopenThreads \
        -L$(VSTASKER_DIR)/Lib/vc140/Osg -losgShadow \
        -L$(VSTASKER_DIR)/Lib/vc140/Osg -losgViewer \
        -L$(VSTASKER_DIR)/Lib/vc140/Osg -losgDB \
        -L$(VSTASKER_DIR)/Lib/vc140/Osg -losgGA \
        -L$(VSTASKER_DIR)/Lib/vc140/Osg -losgText \
        -L$(VSTASKER_DIR)/Lib/vc140/Osg -losgUtil \
        -L$(VSTASKER_DIR)/Lib/vc140/Osg -losgSim \
        -L$(VSTASKER_DIR)/Lib/vc140/Osg -losgParticle \
        -L$(VSTASKER_DIR)/Lib/vc140/Osg -losgAudio

INCLUDEPATH += $(VSTASKER_DIR)/Include/Osg \
               $(VSTASKER_DIR)/Include/Osg/Root \
               $(VSTASKER_DIR)/Include/Osg/Audio

SOURCES += $(VSTASKER_DIR)/Src/Osg/osg_setup.cpp \
            $(VSTASKER_DIR)/Runtime/Qt/vt_qt osg.cpp
```

Once this is done, you will be able to compile and link, but nothing will appear because you will need to call three functions (defined in `vt_qt osg.cpp`)

In your Qt main window class constructor, where you create `vt_rtc` and just after `autorun()` function, add the following call:

```
qtOsgInit(vt_rtc);
```

You will need to add the necessary include on top of the file:

```
#include "../runtime/qt/vt_qt osg.h"
```

Then, in the loop timer, add after the `vt_rtc->tic()` the following call:

## Qt + OSG

```
qtOsgTic(vt_rtc);
```

and finally, in the exit/destructor part, where you delete `vt_rtc`, clean the OSG thread properly:

```
qtOsgExit(vt_rtc); // put that before delete vt_rtc !
```

That's it. You now have the OSG window displayed and ready to use.

# **Copyright**

vsTASKER software is the property of VirtualSim Sarl based in Nice, France.

vsTASKER is copyrighted by VirtualSim Sarl - All rights reserved.

## **IMPORTANT - READ CAREFULLY**

This license statement and limited warranty constitutes a legal agreement ("License Agreement") between you ("Licensee", either as an individual or a single entity) and VirtualSim Sarl. ("Vendor"), for the software product vsTASKER ("Software") of which VirtualSim Sarl. is the copyright holder.

**BY INSTALLING, COPYING, OR OTHERWISE USING THE SOFTWARE, YOU AGREE TO BE BOUND BY ALL OF THE TERMS AND CONDITIONS OF THE LICENSE AGREEMENT.**

Upon your acceptance of the terms and conditions of the License Agreement, VirtualSim Sarl. grants you the right to use the Software in the manner provided below.

If you do not accept the terms and conditions of the License Agreement, you are to promptly delete each and any copy of the Software from your computer(s).

The Vendor reserves the right to license the same Software to other individuals or entities under a different license agreement.

After accepting this license agreement, the Licensee is permitted to use the Software under the terms of this agreement for no more than the number of days allowed by the Evaluation license, without payment to the Vendor.

The Permanent license bears the name of the licensed person or entity, the computer LAN card number and is not transferable to any other party or computer. Pricing and availability is subject to change without prior notice. The Licensee can consult the most recent pricing information at [sales@virtualsim.com](mailto:sales@virtualsim.com).

The Software is provided "as is". In no event shall the Vendor or any of his affiliates be liable for any consequential, special, incidental or indirect damages of any kind arising out of the delivery, performance or use of this Software, to the maximum extent permitted by applicable law. While the Software has been developed with great care, it is not possible to warrant that the Software is error free. The Software is not designed or intended to be used in any activity that may cause personal injury, death or any other severe damage or loss.

When errors are found in the Software, the Vendor will release a new version of the Software that no longer contains those errors a reasonable amount of time after the Vendor is given an accurate description of those errors. Which amount of time is reasonable will depend on the complexity and severity of the errors. The Vendor will mention the release at <http://www.virtualsim.com> and, at the Vendor's option, directly contact the Licensee to announce the new release.

New releases are free to download for Licensees under valid Maintenance contract. Licensees that want to download the new release and who are not under Maintenance contract might pay either the full price of the new release or the Maintenance fees he would have paid from the time he bought the product until the date of the new release issue, whichever comes cheaper.

You must not attempt to reverse compile, modify, translate or disassemble the Software in whole or in part. You must not run the Software under a debugger or similar tool allowing you to inspect the inner workings of the Software.

The Software remains the exclusive property of the Vendor. Any Licensee which fully complies with the terms in this license agreement may use it according to the terms of this license agreement. You must not give copies of the Software or your license key to other persons or entities. You must not transfer the Software or your license key to another person or entity.

## **Copyright**

You must also take reasonable steps to prevent any third party from copying the software from one of your machines without your permission.

You may distribute the demo version (and the demo version only) of the Software that is available for public download at <http://www.virtualsim.com> at the moment that you do distribute it, on the condition that you do this by making identical copies of the downloaded file(s). Public download means any file that can be downloaded by browsing to <http://www.virtualsim.com> and navigating through the links visible on the page, without the use of any password or identification that you may type in or that may be automatically supplied by your browser if you have typed it in before.

You must not ask payment for the act of distributing the demo version of the Software.

The Vendor reserves the right to revoke your license if you violate any or all of the terms of this license agreement, without prior notice.

All license requests must be sent to support@virtualsim.com

Copyright 2004 - 2021

