Configure multiplayer- and AI vehicles

Table Of Contents

Multiplayer and AI vehicles, i.e. third-party vehicles, are usually set in the vehicle configuration of the "real" vehicle.

1 Model

Visu-Flags blah blah.

2 Script

I will explain the principle, according to which the scripts are made fit for the AI and the multiplayer, first of all using the example of the multiplayer.

Imagine two brothers, an older and a younger one. The older one was at a sports course and got explained exactly which exercises he should do to strengthen his back (because he is sitting at the PC all day long and only plays LOTUS 😉). The younger one was not at the sports course, but sits at the PC just as much and plays LOTUS. They now want to save the costs for a second course. Because the younger one also knows the exercises but does not know when to do which and how often, they simply do the exercises together and the older one always tells the younger one which exercise to do next, how long and how often.

I think this example explains very well how the Multiplayer works internally in LOTUS.

2.1 Problem definition

The goal of multiplayer is that colleagues see, hear and experience their own train in the same way that I see, hear and experience it myself. Of course, the most important thing is that it is the right train (i.e. the ContentIDs of the individual vehicles are transmitted) and that it is in the same place (i.e. this is also transmitted). At the colleagues, a train is now spun with the transmitting ContentIDs, which is always "pushed" to the position where I am at the moment with the position data.

We call this train the "multiplayer (MP) train" or "receiving train". The own train we call the "user train" or "sending train".

However, it would be very boring if colleagues experienced my train completely unlit and silent. So the MP train should be animated and "sounded" just like my user train.

2.2 Unsuitable approach

The obvious thing to do now would be to send all public variables of the user train to the colleague, so that the MP train makes the same animations and noises as the user train in my case.

But this method has several disadvantages:

- The amount of data to be transmitted: A tram or a bus has hundreds of variables, depending on its complexity. Almost all of them have internal functions and are not needed for the external appearance and sound.
- Poor update rate: The transmission speed of the data packets is limited and the update interval is significantly lower than the frame rate. A light that flashes quickly in the user car (e.g. the outer door closing warning lights on the GT6N) will therefore only flash very irregularly and much more slowly in the MP car. Animations would only be updated when a new variable value arrives again, and therefore run more or less jerky.
- Incompatible to the AI: If the train should be controlled by the AI instead of the multiplayer, the AI would have to "guess" which variable has which meaning. So it would be simply impossible to switch on the train's headlights at night, for example. We don't even have to talk about the door controller...

For the reasons mentioned above, the "brother model" is therefore used:

2.3 Appropriate approach and procedure in LOTUS

The user train is the older brother. It runs the normal script with all variables and procedures. The MP train is the younger brother. He has quite some skills and knows how to calculate and animate/sound one or the other thing in detail. So his vehicles also have scripts, but they only contain the information how certain things (e.g. opening doors) are processed. But you cannot decide when and why something happens.

For this purpose the scripts of the user vehicles write into certain standardized variables when they do certain things that are important for the MP-vehicles or which state certain functions have. The same variables can now be used by the MP-vehicles after the transmission, so that their script can interpret these "instructions".

The list of these special variables is fixed. Therefore on the one hand the transmission via internet is uniform and limited to a minimum and on the other hand they can also be set by the AI so that the vehicles controlled by the AI also contain appropriate animations and sounds.

Since this system is similar to a remote control, the term RC (remote control) is also used. The list of RC variables can be found here: Lexicon link.

From the described RC system it follows that the script of a vehicle behaves differently depending on whether it is in normal mode (as user vehicle) or in RC mode (as MP-/KI vehicle). Therefore there are...

- ... a different procedure call if the car is in receive/RC mode: Instead of SimStep, SimStep_RC is called (see link)
- ... a variable: RC_Active, which is set to true if the car is running in RC mode. If it is false, then it is in transmit, i.e. normal mode.

Last but not least you have to take care that the variables are always transmitted for the whole train. This saves transmission capacity and makes it easier for the AI. This results in some tricks with certain variables, but these are explained for each variable in the list of RC variables.

2.4 Recommendation

A separate include file should be created, which processes the SimStep_RC procedure. The corresponding include command should be the last one after all other include commands so that the SimStep_RC procedure can access all previous procedures and variables.

2.5 Example

The warning bell of the GT6N shall serve as an example here:

In the normal SimStep procedure it is first determined whether the corresponding environmental conditions occur. If this is the case, a 1 is written into the provided variable `RC_Belling`, otherwise a 0:

Code

```
1. if (Cockpit_A.Active and Cockpit_A.Ts_Klingel.Value) or ((TractionAndBrake_FastBrake or (Cockpit_A.Active and Cockpit_A.TS_MgBremse.Value)) and (not TractionAndBrake_VehicleStopped)) then
2. RC_Belling := 1
3. else
4. RC_Belling := 0;
```

Then the procedure `SetKlingel` is called, which is structured as follows and is responsible for controlling the sound variables:

Code

```
1. procedure SetKlingel;
2. begin
3. if RC_Belling <> Cockpit_Klingel_Prev then
4. begin
5. if RC_Belling = 1 then
6. Snd_Klingel_Loop := 1
7. else
8. begin
9. Snd_Klingel_Loop := -1;
10. Snd_Klingel_End := 1;
11. end;
12. end;
13. Cockpit_Klingel_Prev := RC_Belling;
14. end;
```

Display More

In this way, when the bell button Cockpit_A.Ts_Klingel.Value is pressed in user-vehicle mode, the bell sound is processed - the bell sounds at the user - and the variable RC_Belling is set to 1. This value is now sent via network connection to the colleagues, where the same vehicle (script) works in MP vehicle mode. There the variable RC_Belling is set to 1 and SimStep_RC is called (which is in the additional RC include file). This file looks like this:

Code

```
1. procedure SimStep_RC;
2. begin
3. ...
4. SetKlingel;
5. ...
6. end;
```

Since SetKlingel checks RC_Belling, the MP train will now also ring - and your colleagues can hear this!

In summary, this example shows concretely that our goals have been achieved:

- Minimal data effort: Only one variable has to be transmitted
- No synchronization problem: Since SetKlingel is also processed locally on the MP vehicle, the two associated sounds, which are time-critically synchronized, are still synchronized correctly. And if the headlights should also flash quickly, this would also be played back correctly (because this is also calculated locally).
- Compatible to the AI: If the AI "decides" that the train should ring, it sets "RC_Belling" to 1 and the AI train rings - and if desired the headlights will blink!