# Software assembly and open standards for driving simulation

**Nicolas Filliard** [1], **Emmanuel Icart** [2], **Jean-Luc Martinez** [1], **Sébastien Gérin** [1], **Frédéric Mérienne**[1], **Andras Kemeny**[1,2]

[1] Arts et Métiers Paristech, Le2i, CNRS, Institut Image,
2 rue Thomas Dumorey - Chalon-sur-Saône, France

[2] Renault, Technical Centre for Simulation

1 avenue du Golf - 78288 Guyancourt

nicolas.filliard@gmail.com, emmanuel.icart@renault.com,
jean-luc.martinez@cluny.ensam.fr, sebastien.gerin@cluny.ensam.fr,
frederic.merienne@ensam.eu, andras.kemeny@renault.com

***Abstract*** *– Driving simulation systems involve a combination of different computation codes. Although some of these modules are application-specific, their majority is reusable and state-of-the-art implementations are readily available in the open source community. This study investigates whether these open source libraries can combine to build a driving simulation application with reasonable performance. To this end, a component-oriented architecture is proposed, in which modules encapsulate relevant libraries behind a standard interface and exchange simulation data through a message passing interface. By integrating a render engine, a physics library and a simple vehicle dynamics model, we were able to rapidly build a functional minimal simulation application supporting distributed execution over a cluster of computers. As this architecture allows the transparent modification of module code and simplifies the addition of new modules, this kernel represents the foundations of an extensible and reconfigurable open source system dedicated to driving simulation. Details on this kernel application and ongoing development of this platform can found at http://open-s.sourceforge.net.*

***Résumé*** *- Les logiciels de simulation de conduite reposent sur une combinaison de différents codes de calculs. Bien qu'une partie de ces modules soit extrêmement dépendante d'un usage particulier, leur majorité est réutilisable et certaines implémentations de pointe sont disponibles dans la communauté du logiciel libre. Cette étude vise à déterminer s'il est possible de combiner ces bibliothèques libres afin de construire une application de simulation de conduite atteignant de raisonnables performances. A cette fin, nous proposons une architecture orientée composant, selon laquelle ces bibliothèques sont encapsulées dans des modules s'échangeant des données relatives à la simulation au travers d'une interface d'échange de messages. En intégrant à cette architecture un moteur graphique, une bibliothèque de simulation de* *physique et un simple modèle de dynamique de véhicule, nous avons pu rapidement mettre en place une application de simulation minimale, pouvant s'exécuter de manière distribuée sur un cluster d'ordinateurs. Cette architecture permettant de modifier le code d'un module de manière transparente et simplifiant l'ajout de nouveaux modules, ce noyau constitue la base d'un logiciel libre extensible et polymorphique dédié à la simulation de conduite dont les détails peuvent être consultés sur le site : http://open-s.sourceforge.net.*

## Introduction

As suggested a decade ago by the initial authors of VRJuggler, an open-source integration platform for virtual reality, VR developers should "concentrate on the worlds they want to create and not the systems on which they run" [22]. The same remark applies to driving simulation. Building a driving simulation software system implies indeed a complex interplay of different modules based on very different technologies and requires a wide variety of technical skills to build a complete system including image rendering, physics simulation, vehicle dynamics modeling, acquisition of drivers' commands, sound generation, traffic simulation, etc.

Although applications of driving simulation may vary and require specific functionalities, the majority of these components follows exactly the same standardized specifications across implementations and could be reused. In particular, state-of-the-art software libraries are available in the *open source* community to handle graphics rendering, dynamics models, inter-process communication, scenario programming, etc. [1-9]. Yet, these well-known open source libraries have never been assembled to build a complete driving simulator. Particularly, questions about their compatibility and the expected overall performance remain undocumented.

A major interest of studying these solutions is to reduce development costs by taking advantage of mature, often cross-platform, open source projects, which benefit from evolutions required in connected communities (e.g. the game industry for visual rendering). Therefore, this approach enables developers of driving simulators to concentrate their efforts on advanced features or specific to industrial applications. Notably, a number of licenses under which open source libraries are distributed like LGPL (Lesser GNU Public License) [11] allow their use in proprietary products. Moreover, using open source software for driving simulation is particularly adapted to experimental research studies which often require specific adaptation to fit the particular needs of their protocols.

### Purpose of the current work

After a brief overview of relevant open-source libraries, we intend to show in this article how these elements can combine to produce a driving simulation application with advanced performance. We present in this article a functioning simulator constructed from a selection of libraries encapsulated into elementary modules. The execution of this application can be distributed over a cluster for higher performance. The underlying architecture facilitates the addition of new modules and transparent code modification. Inter-module communication layer is based on message passing implemented using a MPI-2[3] standard compliant library. This kernel includes a state-of-the-art real-time image generator used in game design displaying a high quality environment imported from ordinary authoring tools (in our case, Autodesk® Maya®, www.autodesk.com), and a control module to drive a car through the environment.

## Software tools for driving simulation and virtual reality

Proprietary simulation products exist such as Oktal's SCANeRtm simulation engine (http://www.scanersimulation.com/), but an alternative open-source equivalent is lacking. Open-source racing games, like TORCS [16], potentially contain all the elementary features required to build a driving simulator, but are hardly re-usable or customizable. Moreover, such systems are often missing support of standard file formats for importing data from usual 3D authoring tools.

Recent surveys have shown the existence of several lower levels for the development of general VR applications which were classified as application programming interfaces (API), frameworks or platforms [18]. APIs are libraries of methods that abstract lower-level resources. Relevant APIs for driving simulation include primarily graphics render engines and/or scene graph management libraries otherwise used in scientific visualization application or game development such as OGRE [1], Irrlicht [2], OpenSceneGraph [3] or OpenSG [4], this latter being additionally able to transparently manage parallel rendering for multi-channel displays. Physics engines handle collisions and are an underlying layer of car dynamics model. ODE [7] and Bullet [8] are two widely supported engines. Proprietary products are also available for free development like nVidia® PhysX® [15] library which enable GPU-accelerated physics simulation. Concerning sound generation OpenAL is a major open source reference [6]. Scripting language such as LUA [9] can also be used for dynamic addition of logic and scenarii in the scenes. Eventually, communication APIs are used to bind together these modules and distribute their execution over computer clusters. MPI standard compliant libraries are suited for this task, such as MPICH [5].

VR development frameworks such as VRJuggler, Delta3D, OpenMASK, OpenSpace3D or FlowVR [18][19] integrate a selection of such APIs and provide a unified development interface. Only low level functionalities are provided and building a driving simulation application would require to program additional custom features, like acquisition of data from steering-wheels or car dynamics model. Moreover, using these integrated frameworks implies to accept the underlying selection of APIs and may have several drawbacks. For instance, FlowVR is not cross-platform, Delta3D does not support cluster management and VRJuggler does not support Microsoft Direct3D rendering. Nevertheless, they guarantee the interoperability between heterogeneous libraries with a controlled

[3] MPI: Message Passing Interface. MPI is a normalized interface for managing message exchanges between processes, initially developed for high performance parallel computing [17].

level of performance. These frameworks will not be use in this study in order to concentrate on selection and testing of open source APIs. Yet, the possibility of importing our application in one of these frameworks in the future is not excluded.

In conclusion, open source libraries provide a set of elementary tools that could enter in the composition of a driving simulation application. However, they are rarely integrated in such a finished product. They are mainly currently used in basic immersive visualization platform or integrated games. Building a useable driving simulator from these elements requires an extensive study of their compatibility, of their overall performance when combined and of their ability to import data from most 3D authoring tools. The modularity and the possibility to modify each part of the simulator is also a crucial point.

## Architecture

### Component-oriented design

The proposed architecture follows a component-based approach, which is suited to integrating heterogeneous modules handling parallel computations. Every specific library is encapsulated in a module providing a standard interface that abstracts its functioning. Although this approach adds several communication steps between modules, practical use in the development of other VR platforms has shown that the additional computation overhead remains negligible. Moreover, this design reduces code coupling between modules, favors code reuse and keeps evolution of the code localized, which are critical requirements in the design phase of such composite application. Therefore, the system can easily evolve with newer technologies due its minimal dependency on particular libraries. Notably, modules based on open source libraries can be transparently replaced by in-house developments or proprietary libraries, provided that a SDK (Software Development Kit) is available.

In the proposed version of the software, modules consist in standalone operating system processes, coordinated using MPICH2 [5], a message passing interface following the MPI-2 standard (see footnote 1). The use of an MPI-based inter-module communication layer enables the transparent execution of the simulation software on a variety of hardware architecture ranging from Ethernet-based computer clusters to multi-CPU computers. MPICH2 benefits from an optimized communication channel called *Nemesis* which accelerates communication between processes executed on a single computing node and which supports efficient shared-memory communication. Moreover, MPICH2 has a widely portable implementation and supports different operating systems including Linux and Microsoft® Windows®.

### Kernel-based architecture

The different modules are organized according to a star-shaped design pattern (Figure 1): every module is connected to a central kernel, which manages a database containing all simulation data (e.g. car position, viewpoint, steering wheel angle). Modules can update these data or receive their last value upon request. This architecture is intended to reduce inter-module execution coupling and to allow the use of heterogeneous module execution frequencies. It also minimizes consequences of module execution failures and facilitates subsequent recovery. This loose coupling enable isolated evolution of modules communication interface, the only constraint being that each module updates and reads the correct shared variables. Moreover, such a centralized data management ensures the coherency of simulation in the whole application and simplifies the implementation of a monitoring tool for message exchanges and data accesses on the server (e.g. for debugging purpose).

This architecture has been preferred over *point-to-point* architecture which arranges modules as a data pipeline, as proposed for instance in FlowVR. Although this latter approach would have optimized inter-module communication speed and simplified inter-module synchronization, the resulting software would have been less robust to a sudden communication loss and a more rigid normalization of messages would have been necessary implying constraints in the evolution of modules.

However, the kernel-based architecture has two main inconveniences that impose constraints on the server dimensioning. First, the central server must be the fastest running process to ensure a correct overall performance. Secondly, central server being busy to handle each incoming messages, its best execution frequencies might drop when the number of modules increases.

## Current state of the software

A basic functional version of this software has been implemented, including four main modules, integrated using a communication layer based on MPICH-2 to enable distributed execution over a computer cluster:

- the visualization module that requests the position and orientation of the camera to display the virtual scene from the driver’s viewpoint,
- the vehicle dynamics module that also handles keyboard inputs which generates the car trajectory from the driving commands,
- a camera manager which transforms car position data into camera position,
- the central server which stores and distributes to the different modules the car and camera positions.

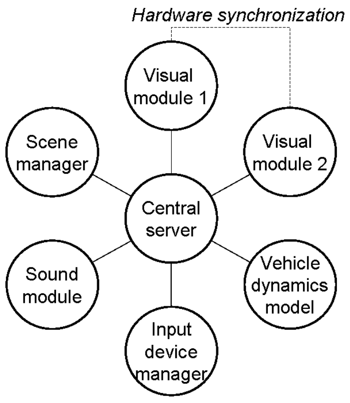
The resulting application consists in a simple functioning driving simulator.

### Interfacing modules and the central server using message passing

The central simulation is primarily a database management system. It stores and distributes upon request the current state of a set of variables. Currently, the central server handles only three types of messages. Modules can order the server to overwrite the current value of some simulation data using “update” messages. “data request” messages are used by modules to fetch the current value of simulation data, in response of which the server releases “send data” messages. Every message used in the application is tagged by a unique identifier which must be declared at the beginning of the application on the server and in the corresponding modules. This identifier is processed in server-side mechanism to decode the simulation data embedded in the message and trigger the expected behavior.

## Visual rendering

The visual module displays in a graphical window a view of the virtual environment, as seen from the driver's current viewpoint which is fetched before the rendering of each frame using a "data request" message.



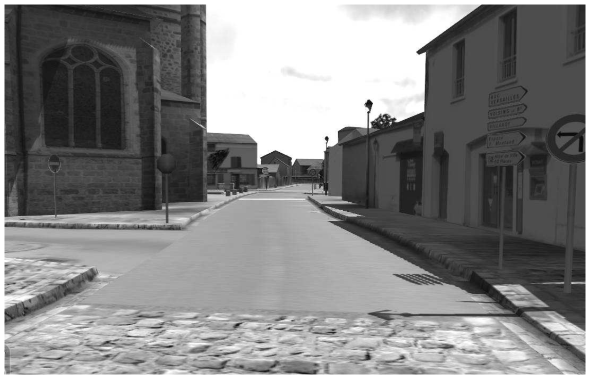


**Figure 1. General module organization**

**Figure 2. Virtual view of Guyancourt rendered with our open source simulation software**

The visual scene is rendered using OGRE [1], an object-oriented open-source 3D render engine written in C++, customizable and extensible through the freely available plugins, enabling for instance OpenGL and DirectX support or bindings to multiple physics engines. Moreover, OGRE benefits from an active community. As currently configured, our visual module supports vertex lighting used to render for instance static ambient occlusion. Advanced projected shadows were also implemented using Parallel Split Shadow Mapping method (PSSM) and rendered using Variance Shadow Mapping technique (VSM) implemented in a pixel shader written in HLSL language (Figure 2). More generally, other visual effects can be implemented depending on the application, the performance of the computers and the features provided by the render engine.

Selecting a particular rendering library also implies constraints on the possible readable database description file formats. OGRE only supports its own native binary format, but a number of conversion plugins and exporters exist for main 3D authoring tools, including open source software like Blender. For the purpose of our study, a visual database modeled in Autodesk® Maya® used in typical simulation scenarii at Renault has been exported using one of these plugins.

Eventually, a particularly important aspect visual rendering for driving simulation is the management of multiple visual channels for multi-screens displays. The proposed solution in the current implementation is to launch as many visual processes as rendering channels that are connected to the server.

## Vehicle dynamics module

Vehicle dynamics module takes as input driver commands and calculates the car trajectory using a physics engine. This module sends messages at its own frequency to the central server to update the car position and orientation data. The current implementation uses ODE open-source physics engine and uses the predefined car dynamics model included in the OGREODE open-source library, initially developed for the purpose of binding ODE with OGRE render engine. Therefore, terrain data and car geometry used for calculation must be provided in the OGRE native geometry description format. In this early version, this module is also responsible for drivers' command acquisition through the keyboard. Implementation of this acquisition operation in a separated module and interfacing with more convenient devices like steering wheels is ongoing.

## Defining a new module: the example of the camera manager

Each module essentially consists in an infinite loop sending messages to the central server either to update some simulation data or request their last available value. The camera manager is a very simple module that illustrates this basic mechanism. It transforms the car position and orientation data computed by the car dynamics model into the position and orientation of the viewpoint in the car expressed in the world frame of reference. The following pseudo-code describes this procedure:

```
while not terminate message received from kernel
    request last car position
    perform viewpoint position calculations
    send 'update viewpoint position' message
    loop frequency regulation instructions
end while loop
```

In addition, server-side mechanisms that listen to these messages should also be programmed. However, due to the limited number of types of messages and their stereotyped processing on the server, server-side message management can be easily standardized and declaration at runtime of both new messages and new simulation data should be possible. Therefore, opening a communication channel between the server and a module will require a few additional function calls at the module initialization to register the messages on the server.

This procedure shows that the code of the simulation server and modules can evolve separately with minor interactions, provided that the structure of the simulation database is not modified. Therefore, this architecture eases the management of version between modules.

## Execution and practical evaluation

Modules are executed as distinct operating system processes, coordinated using MPICH2 utilities including daemons which ensure message delivery. Modules can be executed on any remote computing node on which this daemon is installed.

Correct execution of the application requires that the central server loops at least faster than the fastest module. MPI being a conservative interface, messages are indeed processed respectively to their reception order. Therefore, a too slow execution frequency of the central server loop results in a lag in the updating of simulation data on the server. Moreover, filling up the incoming message stack of the server blocks the execution of the message sender and eventually results in an overall slow down of the application.

Although experimental evaluation of the performance of the described implementation cannot be precisely documented at this level, this implementation allowed a real-time driving in a typical driving simulation environment (~ 300.000 polygons) on a single desktop computer (Intel Xeon 3GHz, 1Go RAM, nVidia Quadro FX 4800). The generation of two synchronized displays (1280x1024) has also been observed with the same level of performance, using two distinct computers connected through Ethernet network.

## Discussion

The functional kernel described in the article validates that open source APIs can be used to build a driving simulation application. The component oriented approach and the kernel based architecture allow reducing the interdependency between the codes of modules and imposing only minimal constraints on message format definition. Therefore, the code of the application can be updated by locally modifying part or integrality of a module, without major influences on the rest of the application. This modular architecture is therefore particularly useful for experimenting different assembly of libraries with the objective of finding an optimal combination. It also provides the foundations of a highly reconfigurable system.

Moreover, the proposed application is scalable thanks to the possible transparent distributed execution over a cluster of computers, and includes strategy for efficient use of memory when running on a single machine, enabled by the use of an MPI-2 compliant library as the communication layer. Preliminary test of the current implementation with low end computers allowed indeed interactive driving. The precise assessment of real-time performance of the application should be addressed in a future study.

Finally, concerning the importation of geometrical information for the visual environment, 3D vehicle models and physics calculation (e.g. collisions, car animation), the application is expecting data in the OGRE native geometry format, for which main 3D modelers have exporters. Yet, an improvement of this importation procedure would be to fully support Collada$^{TM}$, a widely used open standard for 3D information exchange [21]. More generally, as the application may encapsulate heterogeneous libraries, each module may require data in its own format, resulting in a potential multiplication of input file formats necessary to run the simulation, which is a drawback. An advanced implementation of the software should also support emerging open standards for driving simulation data description such as OpenDRIVE® [13] and RoadXML© [14].

## Future directions

Future work will focus on further integration of modules particularly input devices, sound management, and more advanced functionalities like a scenario building interface and traffic management. This latter topic will necessitate to study and select formats for road network specifications, among which new open standards, OpenDRIVE® and RoadXML©, are of particular interest.

## Conclusion

This article presented an ongoing work on the design of an extensible, highly configurable and scalable open-source driving simulation software system. The proposed architecture allows an easy interfacing of third-party libraries to take advantage of existing state-of-the-art functionalities of open-source or commercial products. The implementation also allows efficient distributed execution over multiple computation nodes for higher performance.

This software is currently being written but a running version including communication layer and a few fundamental modules necessary to drive a car in a virtual environment is already being published as Open-S project, licensed under LGPL, available for downloading and testing as of end of June 2010 on http://open-s.sourceforge.net.

## Acknowledgements

The virtual environment displayed in the figures of this abstract representing the city of Guyancourt was designed and realized by Jean-François Rivière (Renault, Technical Centre for Simulation).

**Keywords:** Simulation design; open-source software; software engineering; component-based architecture; distributed applications

## Bibliography

[1] OGRE – Open Source 3D Graphics Engine, http://www.ogre3d.org

[2] Irrlicht Engine - A free open source 3d engine, http://irrlicht.sourceforge.net

[3] OpenSceneGraph, http://www.openscenegraph.org

[4] OpenSG, http://www.opensg.org

[5] MPICH2: High-performance and Widely Portable Message-Passing Interface (MPI), http://www.mcs.anl.gov/research/projects/mpich2

[6] OpenAL: cross-platform 3D audio API, http://connect.creativelabs.com/openal

[7] ODE – Open Dynamics Engine, http://www.ode.org

[8] Bullet Physics, http://bulletphysics.org

[9] The programming language LUA, http://www.lua.org

[10] FlowVR, http://flowvr.sf.net

[11] GNU Lesser General Public License, http://www.gnu.org/licenses/lgpl.html

[12] COLLADA – Digital Asset and FX Exchange Schema, https://collada.org

[13] OpenDRIVE – managing the road ahead, http://www.opendrive.org

[14] RoadXML – The open format road network, http://www.road-xml.org

[15] nVidia PhysX Physics simulation for developers, http://developer.nvidia.com/object/physx.html

[16] TORCS Driver Simulator. http://torcs.sourceforge.net.

[17] MPI specification, http://www.mpi-forum/docs

[18] Wright T.E. and Madey G.□, A Survey of Technologies for Building Collaborative Virtual Environments, *The International Journal of Virtual Reality*, 2009, 8(1):53-66

[19] Soares L.P., Raffin B. and Jorge J.A., PC Clusters for Virtual Reality, *The International Journal of Virtual Reality*, 2008, 7(1):67-80

[20] Morillo P., Bierbaum A., Hartling P., Fernandez M., Cruz-Neira C., Analyzing the performance of a cluster-based architecture for immersive visualization systems, *Journal of Parallel and Distributed Computing*, 2008, 68(2):221-234

[21] Arnaud R., Parisi T., Developing Web Applications with COLLADA and X3D, White paper, 2007, khronos.org

[22] Bierbaum A., Just C., Hartling P., Meinert K., Baker A., Cruz-Neira C., VR Juggler: A Virtual Platform for Virtual Reality Application Development, *IEEE Virtual Reality Conference 2001 (VR 2001)*, 2001, pp 89-96