A Photorealistic Rendering Infrastructure for Man-in-the-Loop Real-Time Vehicle Simulation

Alessandro Tasora
Dept. of Engineering and Architecture
University of Parma
Parma, Italy
email: alessandro.tasora@unipr.it

Dario Mangoni
Dept. of Engineering and Architecture
University of Parma
Parma, Italy
email: dario.mangoni@unipr.it

***Abstract*— We discuss a software system for high-quality interactive rendering of virtual environments. Such tool embeds a state-of-the-art rendering engine middleware that is capable of rendering environments with high level of detail at interactive frame rates on modern GPUs. The model of the vehicle is defined via a model-based Functional Mock-up Unit that can be generated with an external tool, using the Modelica language.**

Keywords – rendering; real-time; vehicle simulation.

I. INTRODUCTION

Thanks to recent advancements in the field of graphics processing unit (GPU) processors, the last generation of 3D rendering engines provides high frame rates even in case of large scenes with high level of detail and complex surface shaders. This allows the adoption of complex effects − such as global illumination and reflections − with a time budget of 20ms per frame, or less; this satisfies the requirement of >50Hz refresh rate for fluid man-in-the-loop interactive simulations, at the same time providing a realistic rendering quality for the best visual cueing [1]. In the past, high refresh rates were achieved at the cost of limiting the complexity of shading and details, hence failing in the so called "suspension of disbelief" effect, that is welcome in fields like virtual reality and vehicle simulators [2]. In detail, the addition of *ray-tracing cores* on the last generation of GPUs can provide unprecedented quality in renderings because ray tracing algorithms can be used, instead of a conventional rasterized rendering [3]. Ray tracing, also evolved as path tracing, can generate physically exact lighting effects, where conventional real-time renderers had to fake effects like reflections or global illumination in sake of performance.

Many applications that require high-performance real-time rendering, such as video games and simulators, are based on extremely powerful third-party middleware such as Unity, CryEngine or Unreal Engine [4][5]. These tools provide ready-to-use rendering algorithms in exchange for some royalties on the final product, or even for free if used in academic projects. In our project, we decided to use Unreal Engine, mostly because it features a well-documented C++ application program interface and because it is renowned for its unparalleled rendering quality.

Although there are many examples of applications that leverage on these rendering technologies for creating car simulations (videogames about racing being a special case of them), in most cases the model of the vehicle is designed directly inside the authoring tools that are provided by the developers of the rendering solutions - in our case it is the Unreal Editor. Doing so, the application could still implement a GUI that allows a user to adjust simple parameters such as the stiffness of a suspension, but if a vehicle designer needs to change some non-trivial property (such as the topology of a suspension or the type of powertrain), the Unreal Editor must be used and the application must be rebuilt again. However, passing through the Unreal Editor all the time that a change is needed, can slow down the design iterations.

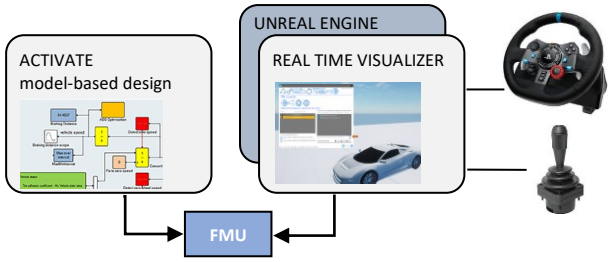

Figure 1. Workflow and software architecture.

In our solution, on the contrary, the user does not need to rebuild the simulator even if radical changes are needed since the vehicle model is separated from the visualization code. In fact, the model of the vehicle is stored into a separated piece of code that always exposes a standardized Application Programming Interface (API), thus allowing for a quick and effective model switching, as shown in Figure 1. This code can be generated by an independent tool − namely an editor for model-based systems − and later loaded in the visualization tool, where the user focuses only on rendering settings and bindings with user inputs.

II. IMPLEMENTATION

Using the workflow that we designed, the physical model of the vehicle is generated with an external tool (namely, Altair Activate), that is capable of creating a

 ISBN: 978-1-61208-898-3

Functional Mockup Unit (FMU) from a model-based description of the vehicle, by leveraging components from both the Modelica Standard Library, containing basic mechanical and electrical components, and from an in-house vehicle-specific library for dynamical vehicle simulations, written in Modelica language [6]. The model, contained in the FMU, is automatically optimized by the underlying Modelica engine in order to offer best performances while retaining the flexibility of block-based graphical user interfaces. Additionally, also a multi-physics approach can be followed to assemble complex systems at a glance: complex vehicles with electric powertrains can be enriched with thermal and dynamical analysis of the system, together with full multi-body suspension geometries; and this without leaving the common Activate user interface.

Figure 2. Example of a wheeled vehicle in a photorealistic environment, including vegetation and atmospheric effects.

Then, we designed a visualization tool based on the Unreal Engine (UE) rendering technology. The performance-critical part of such tool is written in C++ thanks to the API and build toolchain of UE, whereas the graphical user interface (GUI) is built using the Blueprint visual scripting system of UE. Currently, only the Windows platform is supported.

In detail, the visualization tool parses the FMU and performs a run-time linking of the libraries that are contained in the FMU, and that define the functions for the time integration of the dynamical model. In order to bind visualization shapes to moving objects, the tool parses the XML file that is embedded in the FMU and that describes the name of the variables: a hierarchical structure of classes is constructed from that information, so to detect if the FMU was generated from Modelica blocks that represent 3D shapes (the Modelica standard defines *Visualizer* classes to this end). Once shapes are detected, a GUI shows a dialog that allows the user to pick a 3D mesh from the disk, as saved from a CAD, or to associate it to an asset prepared with the Unreal Editor and packaged in a .pak file. The latter option is meant for advanced users: at the cost of requiring the Unreal Editor, it allows additional effects such as the addition of particle effects and sounds, for instance spinning wheels can generate smoke and scratches at the ground, while engines can produce realistic noise. We also provide a method for bidirectional connectivity between the Unreal Blueprint scripts and the FMU variables.

The user can also attach inputs such as steering wheels, joysticks and buttons to FMU variables. Vice versa, output FMU variables can be exported to plots, CSV file logs, GUI and user-designed head-up displays, so that a full Human Machine Interface (HMI) can be implemented and tested in real-time.

Additional GUI panels allow the control of the level of photorealism, enabling depth of field, lens flares and lens bloom, global illumination, motion blur, color grading, etc.

The user can import scenarios designed with the CAD or with Unreal Editor, for example a vehicle can be tested in a virtual city or in a desert land or off road, as in the example of Figure 2. The sky and weather of the imported scenarios can be modified in run time thanks to a real-time atmospheric subsystem that generates sun, moon, stars, sky scattering, clouds and fog.

III. CONCLUSION

We designed a tool that allows the run-time linking of FMU in a visualization framework. This system allows efficient and photorealistic simulations of vehicles of man-in-the-loop type.

ACKNOWLEDGMENT

This work has been partially sponsored by Altair Engineering Inc. We thank Ewald Fischer, Chrysa Nikopoulou, Georgios Ntaountakis, Pranay Kumar, Michael Hoffmann, Filippo Donida, Livio Mariano, Franck Delcroix and others at Altair for testing the beta release of the tool and for reporting bugs and suggestions.

REFERENCES

[1] T. Akenine-Möller, E. Haines, and N. Hoffman, Real-Time Rendering, 4th edition, CRC Press, 2018.

[2] M. Pharr, W. Jakob, and G. Humphreys, Physically Based Rendering: From Theory to Implementation, 3rd edition, Morgan Kaufmann, 2016.

[3] Nvidia RTX platform, https://developer.nvidia.com/rtx [retrieved: July, 2021].

[4] Unity rendering engine, http://www.unity.com [retrieved: July, 2021].

[5] UnrealEngine rendering engine and 3D content creation tool, http://www.unrealengine.com [retrieved: July, 2021].

[6] P. Fritzson, Introduction to Modeling and Simulation of Technical and Physical Systems. Wiley-IEEE Press, 2011.

[7] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 3.3, Wiley, 2014.

 ISBN: 978-1-61208-898-3