ISSN 1346-5597

NII National Institute of Informatics

NII Technical Report

A Short Tutorial of the Scenario Markup Language

Kugamoorthy GAJANANAN
communicated by Prof. Helmut Prendinger

NII-2013-001E
Jan. 2013

A Short Tutorial of the Scenario Markup Language

Kugamoorthy Gajananan

Abstract

We present the Scenario Markup Language (SML), a practical language for authoring realistic traffic situations. This effort is part of a novel framework for conducting controlled driving behavior studies based on multiuser networked 3D virtual environments. SML facilitates the scripting of traffic scenarios for behavioral driver studies in networked multi-user online 3D virtual environments.

I. Introduction

Traffic engineers who focus on the study of driver behavior (e.g., route choice) are interested in using virtual world environments, to extract secondary data sets of natural responses of participants exposed to specific transport situations. Driver responses during these situations are measured to collect reliable behavioral data, which is analyzed to validate some hypotheses (e.g., about a new transport measure). Large scale behavioral studies are required by traffic engineers to properly support these hypotheses. Examples include traffic phenomena like the rubbernecking effect (explained later in Section 3), intersection safety, car following in traffic congestion, among others.

Traffic engineers rarely conduct behavioral studies in the real world, as the tested measures may interfere with real world traffic. Instead, they use web-based surveys which, however, restrict the type of behavioral data collected. In recent years, simulators were built to address several aspects of transport, including traffic flow, driving experience, and pedestrian behavior. To conduct transport behavioral experiments in simulated environments, traffic engineers want to create predictable experiences for human participants. To achieve this, they require realistic scenarios and precisely controlled events [1]. Here, traffic engineers author scenarios that unfold in a virtual world environment, whereby the scenario script orchestrates the simulation run to create predictable experiences for the participants.

To author realistic scenarios for use in virtual world simulations, scenario-based programming handles the interaction between entities. The challenges here are to create believable entity behavior and to orchestrate different entities' behavior into credible, controlled scenarios [2]. A major issue for researchers of scenario languages is the gap between the specification of scenarios in a high level language (by experts) and the technical implementation of scenario specification (by developers).

To address the shortcomings related to scenario authoring of other related approaches (see Section 2), we propose a simple XML based language called Scenario Markup Language (SML) as a practical tool. Using SML, traffic engineers can author complex transport scenarios. Our solution is based on an online virtual world platform, which allows users to be immersed in the virtual world via a graphical self-representation (as avatars), so that users can participate as drivers, pedestrians, or traffic engineers.

II. Related Work - Scenario Authoring Languages

This section reports on work related which focuses on specifying scenarios in a high level language appropriate for non-programmers. In scenario based programming, the challenges are to create believable entity behavior (animation and simulation) and to orchestrate different entities' behavior into credible, controlled scenarios in 3D simulated environments.

First we look at the scripting languages targeted for behavior programming for entities in 3D simulated environments. There has been a family of languages introduced for animation and simulation behaviros of entities in 3D:

- Perlin *et al.* [3] introduced the Improv, which is an action based scripting language to empower computer animators to create behavioral entities which can be controlled by scripts. Using Improv, well-defined actions, which are routines that control an object's degrees of freedom for a period of time, can be created and later invoked in the simulation.
- Conway developed Alice [4], which is programming environment to make 3D animation accessible to novices. Alice provides its own scripting language based on the Python language to control and describe the movements of objects in the environment.
- UnrealEngine [5], which is a game development toolset designed with ease of content creation and programming in mind, provides the UnrealScript language [6] as a powerful, built-in programming language that maps naturally onto the needs and nuances of game programming. The UnrealScript language defines a distinct notion of state and can be used to build autonomous agents
- Unity3D [7],very similar to UnrealEngine, is a software platform that consists of an integrated tool for design and development of 3D real-time simulation content or games. Unity3D allows developers to write simple behavior scripts in JavaScript, C# or Boo.

It is obivious that, except Alice, all of the above mentioned scripting languages are intended for 3D real-time simulation programmers who work on the low level behavior programming.

Secondly, we review what languages are available for authoring scenarios, especially in the traffic domain.

Besides the scripting language of Willemsen *et al.*[8], and the programming style language of Devillers and Donikian [9], other approaches for traffic scenario authoring were developed.

Early work on scenario scripting language by Van Winsum [[10], [11]] propose a text based Scenario Specification Language or SSL which allows the simulation entities to be commanded to perform certain actions. Since then, SSL is further developed as a commercial product at STsoftware [**?**] and SSL is now referred to as StScenario [**?**].

Allen *et al.* [12] also use a textual description of scenarios, whereas others utilized a specialized representation called 'tile' [[13], [14]]. The idea of specifying a series of tiles in a configuration file can be sufficient for simple scenarios, but cannot scale well for complex dynamic and non-deterministic scenarios. Another disadvantage with textual and tile based approaches is that the entire scenario (static and dynamic elements) need to be specified beforehand in texts or tiles.

In this section, we reviewed related work, in detail, that uses different approaches for scenario authoring. We also discussed the limitations of the related approaches. The work presented in this technical report relates directly to the body of work discussed above, and uses many of the ideas and guidelines for authoring scenarios. Our chief goal is to make scenario authoring practical for traffic engineers to specify traffic scenarios.

III. Traffic Scenario Authoring with Scenario Markup Language

SML is a high-level, practical specification language for authoring traffic scenarios. A preliminary version of the basic concepts of the SML was already introduced in Gajananan *et al.* [15]. In this section, we describe the core tagging structures of the SML language, which are used to specify the target traffic scenario.

Note that the usage of the word ‘scenario’ sometimes refers to both the static structure of the virtual environment or scene (e.g., road network) and the dynamic characteristics of a simulation (e.g., critical events) [16]. We use scenario (script) to refer to entities (vehicles) in the simulation and the orchestration of their behaviors (interactions) to create a specific situation or event [17].

A. Motivating Traffic Scenario on Rubbernecking Effect

For this work, we choose a traffic accident scenario as a study case which could benefit from the existence of multiple drivers in driving simulators. As for the accident scenario, we specifically mean a situation in which a following car collides with a leading car from back. The chosen scenario helps traffic engineers to investigate how real drivers react to each other when they perceive the accident situation (i.e., how drivers change their operational driving behavior at the incident site). The scenario like an accident situation in which drivers’ interactions are too dangerous or unpractical to be studied in the real world.

This particular scenario is motivated by the traffic phenomenon called rubbernecking, whereby drivers tend to crane their neck in order to get a better view of a nearby car crash (see Fig. 1). This phenomenon can cause traffic congestion to surge, for drivers may become distracted and drastically change their rate of travel, which in turn may cause traffic congestion on the opposite side of the road where the accident happend [18]. Traffic engineers identified this traffic phenomenon as a leading cause of traffic congestion and secondary accidents. They claim that drivers looking at vehicle crashes and roadside traffic incidents cause the rubbernecking effect. We use this traffic accident scenario as a running example to describe the tagging structures of the Scenario Markup Language (see the following section).

B. Scenario Markup Language Concepts

We conceptualize a scenario as the script that specifies the behaviors (defined by the interests of the scenario author) of entities, and how those behaviors are orchestrated, by creating, destroying or modifying characteristics of

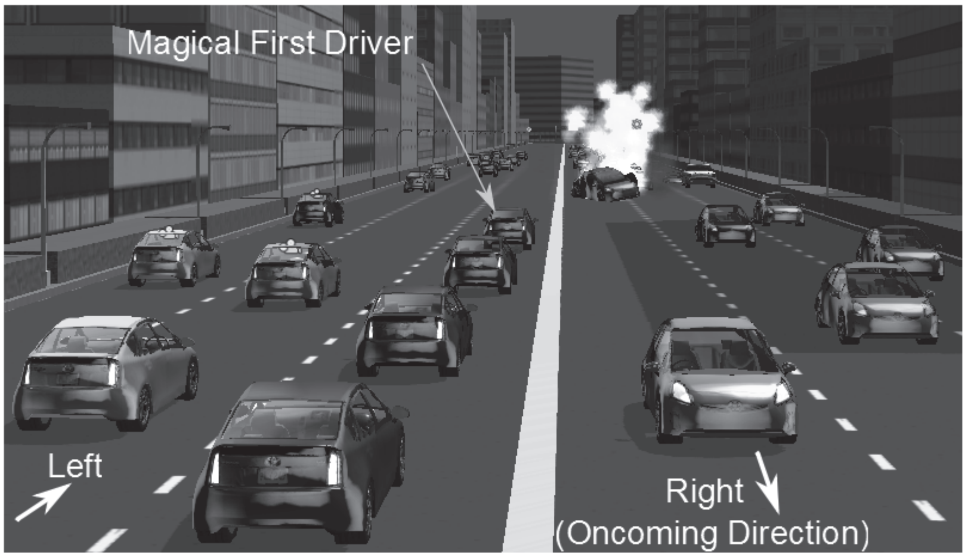

Fig. 1. Illustration of a rubbernecking scenario. In this example, an accident has happened on the right side of a road (oncoming direction). The first driver on the left side (reference side), is the one who first perceives the accident and, likely, initiates the formation of traffic jam.

entities and coordinating their actions, to produce experiences for human users immersed in a 3D virtual environment. In this case, an entity refers to a materialized or embodied virtual object (e.g., a car) or to a virtual character represented as a human avatar (a synthetic human user or a bot). Users who are participants of a behavioral study may interact with one or more entities in the virtual world during a simulation run designated for the behavioral study.

A behavioral entity perceives its environment, makes decisions, performs actions and has an interface for communication. Our conceptualization of defining the behavior of entities is similar to the definition above.

1) SML Tagging Structure: As shown in the Fig. 2, the root element of a scenario script is SML. The tagging structure *<SML></SML>* contains the header and body parts. As in Fig. 3, the header part of an SML script is composed of the tagging structures such as ‘entities’ — the entities involved in a scenario, ‘user’ — the user referenced in a scenario. Fig. 4 shows the body part of an SML script that contains the tagging structure *<Scenario></Scenario>* all other tagging structures, in particular ‘director’ — the portion of script that orchestrates certain behaviors of specified entities, and ‘events’ — generated as an outcome of executing the scenario, ‘behaviors’ — a set of interesting behaviors for the entities defined above, from a scenario point of view. We assume that entities will have their autonomous behaviors implemented, will behave accordingly mostly except the time the behaviors defined in a scenarios are executed. In addition, we implement a set of interface methods to

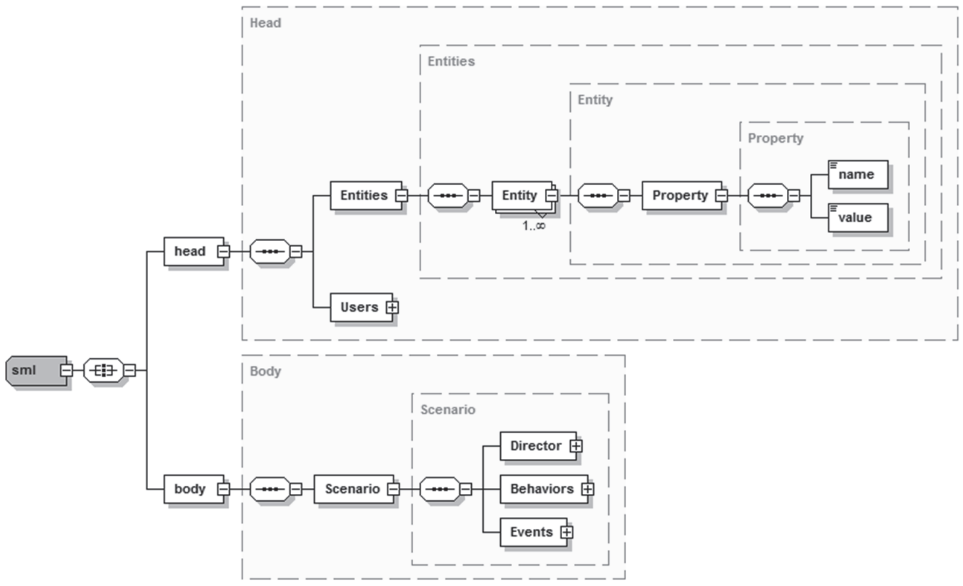

Fig. 2. The structrual diagram that provides an overall structure of an SML script for scenario, two main parts: the head and the body and their abstract content.

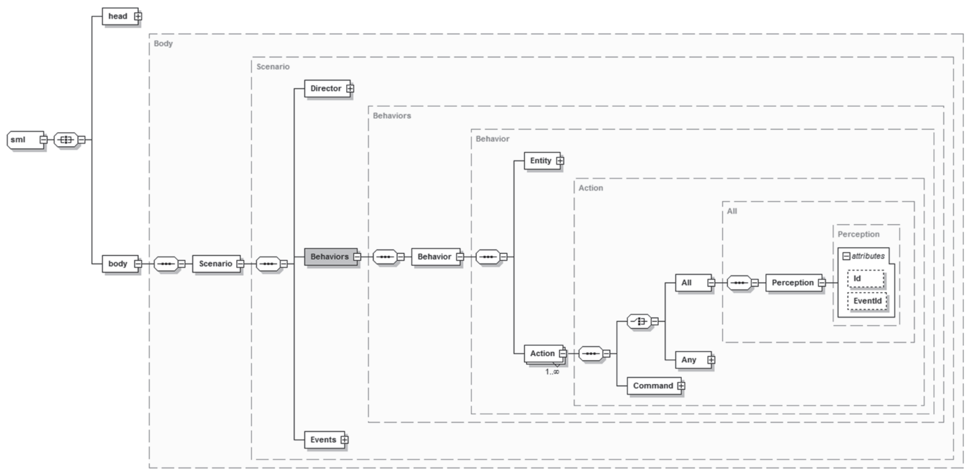

Fig. 3. The structrual diagram that provides an overall structure of the header part of an SML script for scenario

direct or command an entity from a scenario.

2) SML Entity: For the accident scenario, we define two car entities along with their attributes and properties (see Fig. 5). We identify each entity by its id. The ids of entities are assigned once the candidates are chosen from the simulation space (e.g., cars that are selected to create an accident). Each entity has a type. Currently, the *type* attribute supports 'Vehicle', which was sufficient for the scenarios implemented now. It can be extended to other types of entities (e.g., pedestrian), once they are implemented in the simulation. To define the properties of an

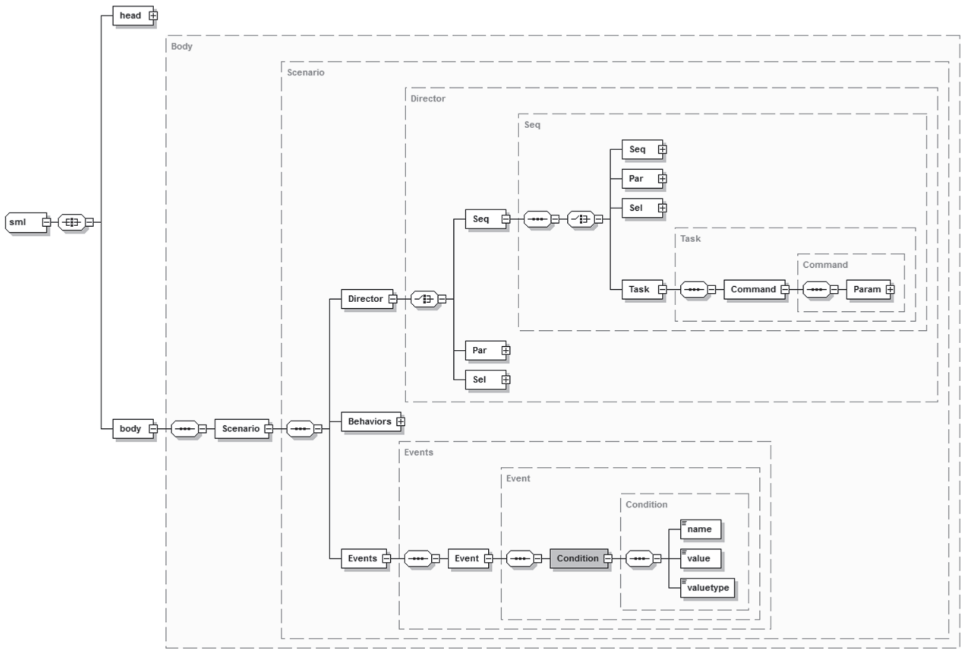

Fig. 4. The structrual diagramthat provides an overall structure of the body part of an SML script for scenario

entity, we specify an element called *Property*. Using this tag inside the *Entity* structure, a script author can define the information to initialize an entity for a scenario.

An SML script lets a script author freely define the properties for an entity. However he/she has to ensure that the component responsible for the entity supports the specified properties. In the example script shown in Fig. 5, we show some example properties that are currently supported for each type of entities used in our simulation. For a car entity, we define (a) *desiredspeed* that specifies the desired speed, (b) *desiredaccel* that specifies the desired acceleration and, (c) *scenarioroleloc* that specifies what location the entity has to take initially in the scenario execution.

Our target environment has to handle multiple users simultaneously. We decided that there has to be at least one *User* as the focal point of a scenario, that is, the experimental condition is to be reproduced from the viewpoint of one specified user. This user is identified as shown in Fig. 6.

3) SML Event: One can define a set of events (e.g., an accident) in a scenario specification to create interesting experiences for users.The event definition specifies the conditions that decide when the event is triggered, as shown in Fig. 7. The conditions for triggering events can be based on an interaction of a human participant with an entity, or the environment, or of entities with others. The conditions can also be based on the changes in the simulation

```
<entity name = "AI Car A" id=  "" type = "Vehicle">
   <property name = "desiredspeed" value = "40m/s"/>
   <property name = "desiredaccel" value = "1.6m/s2"/>
   <property name = "scenariorole" value = "Follow_Car"/>
   <property name = "scenarioroleloc" value = "ROLE_LOCATION_A"/>
</entity>
<entity name = "AI Car B" id=  ""  type = "Vehicle" >
   <property name = "desiredspeed" value = "30m/s"/>
   <property name = "desiredaccel" value = "1.5m/s2"/>
   <property name = "scenariorole" value = "Lead_Car"/>
   <property name = "scenarioroleloc" value = "ROLE_LOCATION_B"/>
</entity>
<entity name = "VMS" id=  "VMS_A"  type = "MessageSign" >
   <property name = "updatetime" value = "5min"/>
   <property name = "location" value = "Intersection_I1"/>
</entity>
```

Fig. 5.   Snippet of an SML script that specifies two car entities and a variable message sign.

```
<user name = TestUser01">
   <property name = "usecarid" value = "U1"/>
   <property name = "scenariorole" value = "Reference_User"/>
</user>
```

Fig. 6.   Snippet of an SML script that specifies the reference user.

state, or a timer or the changes that occur in the external applications such as a traffic simulation application.

4) SML Behavior, Perception with its Container, Action, Command: For each behavior (e.g., ManageMessages), we specify one or more actions (e.g., UpdateMessage) inside each behavior specification as shown in Fig. 8. Each action specifies a perception container which has a set of perceptions defined in it. A behavioral entity perceives its environment (e.g., Accident Event) via the perceptions. Each perception is contingent upon an event, which means that a perception is the minimum unit of event handling from the script perspective. Hence the perception is activated when the corresponding event is triggered during a simulation. This may lead to the activation of the perception

```
<event>
    <condition>
        <var name = "eventid" valueType = "string" value= "Accident_Event_01"/>
        <var name = "collider" valueType = "string" value= "AI Car B"/>
        <var name = "collidedwith" valueType = "string" value = "AI Car A"/>
        <var name = "location" valueType = "vector3" value = "ACCIDENT_LOCATION"/>
   </condition>
</event>
```

Fig. 7.   An example SML script that specifies an accident event that occurred in the 3D virtual environment as a result of executing a scenario.

```
<behavior  ref = "VMS_A"  id = "ManageMessages">
    <action id = "UpdateMessage">
        <perception id = "P3" eventid = "Accident_Event_01"/>
        <command id = "Update">
            <param name = "Msg" value = "Reduce Speed, Accident Ahead" />
        </command>
    </action>
</behavior>
```

Fig. 8. An example SML script that specifies a behavior for an entity.

container the perception belongs to (decision making aspect of an entity). In turn, this leads to the execution of an entity's action that the perception container belongs to. Therfore a perception acts as a pre-condition for an action.

An action definition can have a set of commands defined in it. Therefore, a behavioral entity performs an action by executing the commands specified for that action. A command is the minimum unit of execution from a scenario and entity perspectives. It carries an instruction or a function call with required parameters for the component that controls the entity. The parameters used with the commands are passed using call-by-value semantics.

5) SML Director Element: In order to orchestrate the behavior of entities involved in a traffic scenario, we propose the concept of a supervisor, specified as *director* in the script. For the director, we specify task elements which contain information about the actions an entity has to perform. Each task defines a command for actions to be executed by an entity. Therefore each task refers to a single entity and is executed by the director of a scenario. A task execution (i.e. sending commands to entities) is performed via the communication interface available for each entity.

For example, as shown in Fig. 9, we define the director for the scenario script that creates the accident situation. In this example, we specify two tasks for the director, whereby each task defines a command to be sent to a different car entity: AI (Artificial Intelligence) Car A and AI Car B.

As shown in Fig. 5, the two car entities involved in the scenario are initialized with a property called 'scenarioroleloc' that specifies what role the entity has to take initially in the scenario execution. In this example, AI Car A takes the 'leading role' hence its 'scenarioroleloc' is specified to be in front of that of AI Car B, which takes the 'following role'. When these car entities successfully reach their respective 'scenarioroleloc', the director executes its task ('MoveToTargetLocation') to create an accident at the specified target location during a simulation run. This operation sends a command to the component that controls the car entity with the parameters specifying the

```
<director>
   <seq>
      <task  id= "MoveToTargetLocation">
        <command>
          <param  name= "entityid" value = ""/>
          <param  name= "entityname" value = "AI Car A"/>
          <param  name= "targetlocation" value = "ACCIDENT_LOCATION"/>
        </command>
      </task>
      <task  id= "MoveToTargetLocation">
        <command>
          <param  name= "entityid" value = ""/>
          <param  name= "entityname" value = "AI Car B"/>
          <param  name= "targetlocation" value = "ACCIDENT_LOCATION"/>
        </command>
      </task>
   </seq>
</director>
```

Fig. 9. An example SML script that specifies the supervisor element (director) of the scenario with its tasks required for the accident scenario.

target accident location. The component controlling the car entity tries to execute the command on the respective car entity so that an accident can be created.

A scenario director can orchestrate the behavior of entities by executing corresponding tasks. This can be done by using the temporal elements of director, such as *seq* (sequential), *par* (parallel) or *sel* (selective). With the *seq* element one can specify that tasks follow immediately one after the other. The *par* element defines that tasks should be executed in parallel. The *sel* element defines that tasks should be executed depending on the condition. These temporal elements enable SML to provide high-level synchronization of the behavior of entities that result in a particular traffic situation. For example, as shown in Fig. 9, the task of the director tasks defined in seq temporal element orchestrates the two car entities' behaviors' by issuing necessary commands, so that an accident, e.g., one in which the following car (AI Car B) hits the leading car (AI Car A), can be created.

Here we wish to note that an empirical evaluation of a scripting tool such as SML is notoriously difficult, as related approaches are sparse and setting up test cases for comparison is unpractical. So, as preliminary empirical evidence on SML's ease-of-use, it was important that our collaborators from the Smart Transport Research Centre at Queensland University of Technology actually used SML to specify the scenario.

References

[1] J. Kearney, P. Willemsen, S. Donikian, and F. Devillers, “Scenario languages for driving simulation,” in *Driving Simulation Conference*, ser. DSC’99, 1999, pp. 377–393. [Online]. Available: http://www.cs.uiowa.edu/~kearney/pubs/dsc99_kearney.pdf

[2] Y. Atir and D. Harel, “Using lscs for scenario authoring in tactical simulators,” in *Proceedings of the 2007 summer computer simulation conference*, ser. SCSC. San Diego, CA, USA: Society for Computer Simulation International, 2007, pp. 437–442. [Online]. Available: http://portal.acm.org/citation.cfm?id=1357910.1357979

[3] K. Perlin and A. Goldberg, “Improv: a system for scripting interactive actors in virtual worlds,” in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH ’96. New York, NY, USA: ACM, 1996, pp. 205–216. [Online]. Available: http://doi.acm.org/10.1145/237170.237258

[4] M. Conway, *Alice: Easy-to-learn 3D Scripting for Novices*. University of Virginia, 1998. [Online]. Available: http://books.google.co.jp/books?id=49lrcgAACAAJ

[5] (2012, August) Unrealengine. [Online]. Available: http://www.UnrealEngine.com/

[6] (2012, August) Unrealscript reference. [Online]. Available: http://udn.epicgames.com/Three/UnrealScriptReference.html

[7] (2011, August) Unity3d. [Online]. Available: http://unity3d.com/

[8] P. Willemsen, “Behavior and scenario modeling for real-time virtual environments,” Ph.D. dissertation, Department of Computer Science, University of Iowa, 2000.

[9] F. Devillers and S. Donikian, “A scenario language to orchestrate virtual world evolution,” in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ser. SCA ’03. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 265–275. [Online]. Available: http://portal.acm.org/citation.cfm?id=846276.846315

[10] W. van Winsum., “Ssl/nsl specification release 1.2,” University of Groningen, TRC, November 1994.

[11] P. van Wolffelaar and W. van Winsum, “Traffic modeling and driving simulation - an integrated approach,” in *In Driving Simulation Conference*, ser. DSC’95, 1995, pp. 235–244.

[12] T. R. R.W.Allen and G. Park, “Scenarios produced by procedural methods for driving research, assessment and training applications,” in *In Driving Simulation Conference*, 2003, p. Paper No. 621.

[13] G. W. Y. Papelis, O. Ahmad, “Developing scenarios to determine effects of driver performance: Techniques for authoring and lessons learned,” in *In Driving Simulation Conference, North America*, 2003.

[14] P. Suresh and R. Mourant, “A tile manager for deploying scenarios in virtual driving environments,” in *In Driving Simulation Conference, North America*, 2005, pp. 21–29.

[15] K. Gajananan, A. Nakasone, H. Prendinger, and M. Miska, “Scenario markup language for authoring behavioral driver studies in 3d virtual worlds,” in *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, sept. 2011, pp. 43 –46.

[16] D. Fisher, M. Rizzo, J. Caird, and J. Lee, *Handbook of Driving Simulation for Engineering, Medicine, and Psychology*. Taylor & Francis, 2010. [Online]. Available: http://books.google.com.au/books?id=lnTA8sT8gkEC

[17] J. Cremer, J. Kearney, and Y. Papelis, “Hcsm: a framework for behavior and scenario control in virtual environments,” *ACM Trans. Model. Comput. Simul.*, vol. 5, no. 3, pp. 242–267, Jul. 1995. [Online]. Available: http://doi.acm.org/10.1145/217853.217857

[18] V. L. Knoop, S. P. Hoogendoorn, and H. J. van Zuylen, “Capacity Reduction at Incidents: Empirical Data Collected from a Helicopter,” *Transportation Research Record*, vol. 2071, pp. 19–25, 2008.