Masterarbeit

Jan Dalski

Design and Implementation of a Generic 3D Visualization Pipeline for Large-Scale Simulations on the MARS Platform

Jan Dalski

Design and Implementation of a Generic 3D Visualization Pipeline for Large-Scale Simulations on the MARS Platform

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Clemen
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 5. Oktober 2018

Jan Dalski

Thema der Arbeit

Design and Implementation of a Generic 3D Visualization Pipeline for Large-Scale Simulations on the MARS Platform

Stichworte

3D-Visualisierung, Multiagenten-Simulation, Ergebnisanalyse, MARS, WebGL

Kurzzusammenfassung

In einer Welt von steigender Komplexität werden Simulationen immer häufiger eingesetzt, um Systemverständnis zu erlangen und fundierte Entscheidungen zu ermöglichen. Insbesondere Multi-Agenten-Simulationen sind in vielen Anwendungsgebieten der Soziologie, Ökologie und Verkehrssimulation stark vertreten. Die 3D-Aufbereitung der Simulationsergebnisse würde einen intuitiven Einblick in die Abläufe gewähren und so ein besseres Verständnis über die Wechselwirkungen des Systems fördern, findet in der Praxis bisher jedoch wenig Anwendung. Diese Arbeit präsentiert eine solche modellunabhängige Visualisierung für die Simulationsplattform MARS. Entwurf und Implementation werden erläutert und es findet eine umfassende Untersuchung des Nutzens für die Anwender und der technischen Leistungsfähigkeit statt.

Title of the paper

Design and Implementation of a Generic 3D Visualization Pipeline for Large-Scale Simulations on the MARS Platform

Keywords

3D visualization, Multi agent simulation, result analysis, MARS, WebGL

Abstract

In a world full of complexity, simulations are frequently used to gain insights into sophisticated systems and to allow profound decisions. Especially multi agent simulations are used extensively in many domains like sociology, ecology and traffic simulation. The use of 3D visualization to present the simulation results may grant an intuitional view on the events taking place in the simulation and the involved entities, yet it is rarely used. This work presents such a model-independent 3D visualization for the modeling-and-simulation platform MARS. It discusses the requirements, design choices and the implementation. Afterwards an analysis of the added benefits for the simulation evaluation and the performance takes place.

Contents

1 Introduction

Our world of today is full of sophisticated and tightly coupled systems. Regardless of whether it is in the social sector, environment protection and ecology or in infrastructural domains like transportation: Complexity is always present and will even increase in the future. This makes it difficult to allow educated guesses about the progress of such systems and how present actions will affect it.

However, exactly this is expected from many key players in leading positions. These individuals have to make sound and often fast decisions, for which it is important to know as much as possible about the direct impact of a decision and its long-term consequences. For such a precise forecast, it is crucial to get insights into a system and and to acquire a good understanding of its correlations.

1.1 Simulations on the Rise

One way of getting this information is to create a model of the system to be examined and all related links. It is essential to choose the right level of abstraction, this means the model has to feature all aspects relevant to investigate the central issue but should be stripped of unimportant or minor influences in order to reduce complexity and disruptive factors (Law (2008), Thiel-Clemen (2013)). With this model, simulations can be made by parametrizing the model with input data and running it for a number of execution steps. Given the accuracy and validity of the model, this process allows to predict the system's development in regard to the input values supplied and thus pointing out their consequences.

When talking about simulations, several fundamentally different approaches exist. *Numerical simulations* are based on mathematical equations that describe a (mostly nonlinear) physical system and are widespread used for weather forecasts and climate models. If it is up to determine the probability a chain of events might occur, the *stochastic* (or *Monte-Carlo*) *simulation* is used. The *dynamical simulation* models the change behavior of a system over time, with a differentiation between *continuous* and *discrete* time representation.

The foundation to this work is the *multi agent-based simulation*, which can be considered as a subset of the discrete simulation. Its underlying simulation model is built using a technique called *agent-based modeling (ABM)*, which employs software agents (autonomously acting programs) to depict real-world entities. Expedited by Grimm and Railsback (2013), the *individual-based modeling (IBM)* is a refinement that says that every simulation individual is represented as an own agent. Illustrative examples for such an agent might be a prey animal in a hunting simulation, a tree in a forest model or a pedestrian in an evacuation scenario.

Contrary to the simulation methods named above, this "bottom-up" modeling approach focuses on describing how the individuals behave and what actions they can undertake in order to influence their environment and other agents (Drogoul et al. (1994)). Taking advantage of the rapid increase of computation power, a large number of agents with complex internal logic can easily be instantiated during runtime (hence *multi* agent simulation). The simulation results arise out of the interplay of all these agents and emergent behavior and dynamic interdependencies can become visible.

1.2 MARS – Multi-Agent Research & Simulation

Emanating from the demand for simulation tooling, many software publishers try to meet these needs by developing use-case oriented and specialized simulation frameworks. Aside from many commercial products distributed by companies, also several academic and community-driven solutions exist. Among them is the MARS (Multi Agent Research & Simulation) research group, situated at the Hamburg University of Applied Sciences (HAW). Primarily consisting of bachelor, master and PhD students, the MARS Group is developing an innovative modeling-and-simulation-as-a-service (MSaaS) platform (the same-titled MARS system, hereinafter just called "MARS") for agent-based simulation (Hüning et al. (2016)).

With its origins in the field of movement ecology simulations (e.g. evacuation scenarios), MARS evolved into a general-purpose simulation platform with a focus on versatility and ease of use. MARS aims to provide the tools for the entire simulation process – from model creation, data import and simulation execution right up to result evaluation. Unified in a web interface, the user can accomplish each of these steps in their web-browser without having the need to install anything. This online workspace also offers cooperation and data integration features and enables the user to build, run and analyze simulations from all over the world.

The necessary power to run such a system is provided by a cloud computation cluster also operated at the HAW. In combination with a simulation engine designed for distribution and scalability, MARS is able to run arbitrary large-scale simulations with millions of simulation entities.

A detailed explanation of the MARS platform and the workflow follows in section 2.4.

1.3 A 3D View into the Simulation World

The process of building and running a simulation is purpose-driven: Whether it is to prove or falsify a claim or to get a better understand of a complex system, the simulation model is built to serve a specific set of goals. As such, it has to be validated and the simulation results need to be investigated in terms of plausibility and peculiarity. MARS, striving to serve as an all-in-one solution, offers a result file download and a web-based, configurable visual analytics dashboard to this end.

Apart from analytical methods, the display of results using 3D techniques could be also a practicable way. In a representation form similar to those of video games or CGI movies, the simulation run can be rendered as a three-dimensional scene with the 3D objects in it depicting simulation entities. The user is equipped with a free-floating camera and video player-style controls, which allow them to freely move around in this virtual world, play back and rewind the simulation steps and look at the individuals at arbitrary angles (Bijl and Boer (2011)).

Though not often used for the scientific evaluation, the 3D visualization may provide a very natural and straightforward way to get a first impression of the events happening in a simulation. This is especially true for multi agent based simulations with geospatial relatedness, e.g. in an ecological domain: An agent can easily be represented by choosing a fitting 3D model; the environment might consist of a heightmap expressing the terrain elevation, optionally enhanced with additional landmarks and textures like street overlays or satellite imagery. It also allows to visualize the actions an agent undertakes (e.g. walk around, chop a tree, hunt an animal) by using computer animation.

This way, a spectator can intuitively grasp complex system behavior by just looking at the scene and how its actors behave. With simulations getting increasingly significant for economical and political planning, such a tool could prove useful to help external stakeholders like decision makers and sales managers to understand the model and thus promoting its acceptance beyond simulation expert circles (Banks and Chwif (2011)).

1.4 Goals

The goal of this work is to provide a 3D visualization as mentioned in 1.3 for the MARS simulation platform. MARS is currently lacking such an option, though the author deems it to be a valuable addition to the portfolio of evaluation tools. In order to be of use the end-user, a set of specifications have to be kept in mind, both regarding *what* the subject of visualization is and *how* it is presented. These demands are detailed in chapter 2.2 and transformed into requirements in chapter 3.1.

Regarding the domain openness of the MARS platform, the visualization should adhere to this generality as well. Large-scale applications are to be expected, so big-data processing and careful result selection algorithms are needed. A smooth integration into the MARS modeling-and-simulation workflow is desired, making it an optional component that shall be usable with minimal configuration overhead. This also implies that the visualization should be accessible directly in the web browser, resulting in the necessity for browser-based 3D rendering technologies and server-to-client streaming techniques. Details on the planned integration and component draft are given in section 3.2.

1.4.1 Hypotheses

The above section mentioned a set of goals that a 3D visualization for a multi-agent simulation system should fulfill and several MARS-related aspects that have to be considered. These goals are now further refined into a set of separate hypotheses. This allows a more precise investigation, if and to what extent the stated goals were achieved.

H1: It is possible to develop a model-independent 3D visualization

With the generality concept of MARS platform and its claim to be applicable in a wide range of domains, the same has to be true for the output visualization. This means that the visualization is not built for a specific simulation model and as such, no prior knowledge of the agent types, terrain characteristics and environments facets is available. Even though a 3D visualization may not be purposeful for every domain so far simulated on MARS (e.g. the immune system model developed by Grundmann (2018)), it should be certainly suitable for any simulation model within the scope of ecology and traffic simulation, which is by now the principal application field of MARS.

H2: The number of agents and terrain extents have no effect on feasibility and performance

Beside the afore-mentioned broad range of applications, another unique selling point of the MARS platform is its scalability. As most notably described and examined in Hüning (2016), the scale of multi-agent simulations is of importance and hence the MARS system was designed to support large amounts of agents and vast terrains. These requirements of course also apply for any result processing step and demand suitable big-data streaming and reduction algorithms to facilitate a web-based exploration of the simulation results.

H3: The usage of a 3D visualization is valuable for the simulation evaluation

The use of 3D visualization tools for simulation evaluation currently only plays a minor role, though the author expects it to be a useful addition to visual analytics dashboards. Especially for decision makers, who often are non-scientists, such a tool may give an opportunity to grasp complex model behavior. In order to prove this assumption, the visualization has to be used by stakeholders, which were not involved in its design and implementation. Also some kind of survey has to be developed in order to make this added value measurable and comparable. Suitable candidates for these tests can be found in (1) the external users of the MARS system to investigate the value for domain specialists, (2) MARS platform developers to prove design comprehensibility and maintainability and (3) outside persons to assess the ease-of-use and the presentation value.

H4: The simulation platform can automatically carry out the majority of the required parameterization

As a consequence of the model independency stated in H1, there is no information on the entities to be visualized available at the time the design and implementation takes place. This requests for a parameterization during setup, where both user-related settings (e.g. initial camera position or 3D models to use) and technical information (agent types, attributes, resource identifiers etc.) have to be specified.

With the 3D visualization being an optional component in the modeling and simulation workflow, it means that this configuration overhead should range from zero (no 3D visualization desired) to only the minimal input necessary. Because the MARS platform is designed as an ecosystem of loosely coupled services, it should possible to obtain all the required information from the model import stages and to provide suitable defaults for the user settings.

1.4.2 Work Not Covered by This Thesis

This section mentions all these aspects and components, which are used throughout the thesis, but are not part of it. With MARS being a complex ecosystem of intercoupled services, a majority of these components provide necessary functionality for the MSaaS workflow but have no relation to the visualization itself. This concerns first and foremost the import pipeline, the parameterization and simulation run setup services and the actual simulation engine. All these parts are given "as is" and are inevitably used, but not further discussed apart from the short presentation about MARS in the next chapter.

The same is true for the cloud-based runtime environment: Whenever necessary for the visualization pipeline, peculiar details are mentioned in the design and implementation chapters, but the underlying hardware, infrastructure, overall cluster orchestration and roll-out of the MARS system is clearly out of scope for this work.

Alongside with the simulation platform, some kind of showcase model is needed to have a visualizable scenario for development and testing. To support the generality claim, two models from different domains are used for that purpose: A savannah ecology model developed by members of the MARS Group together with the ARS AfricaE project (Falge et al. (2012)) and a traffic model which depicts the commuting traffic in the city of Hamburg (developed as a joint project between HAW and Hamburg University). These models are made available for this work; however their design, implementation and contentual significance are not examined.

Each of these models also requires a set of input files (climate measurements, street maps, census data...) and parameterization in order to start a simulation run. Again, the validity of these input data and mappings is not relevant to this thesis, because the models are in this case just a means to an end – to facilitate the development of the 3D visualization. Their exploration and scientific evaluation is up to the respective domain experts.

1.5 Document Structure

After this short introduction into the background and objectives here in chapter 1, the document continues with a more comprehensive overview on the topic, existing solutions, the targeted area of application and an in-depth presentation of the MARS platform in chapter 2.

Once these fundamentals were discussed, chapter 3 turns towards the design of the visualization pipeline. Starting with a requirements analysis for the system to be build, this chapter proceeds with a rough sketch of the overall pipeline and its partitioning into subcomponents. Based on that outline, the implementation is detailed in chapter 4. Divided into upfront configuration, result output and visualization, the individual components and their technical realization are presented.

With the system at this stage being completely built, it is time to conduct some experiments in order to prove the hypotheses. Chapter 5 first describes the common boundary conditions and then outlines several experiments for that purpose. The results of these experiments are presented and subsequently discussed in chapter 6.

Finally, chapter 7 closes this thesis by drawing a conclusion and giving an outlook on possible improvements and extension points for those who may want to continue this work.

2 Material & Methods

This chapter gives a more thorough introduction into the principles of agent-based simulation (ABS), individual-based modeling and (massive-)multi-agent frameworks, upon which this work is based. It discusses why a 3D visualization may be of use, especially in the field of ABS. Beside the advantages, it also lists several key aspects that have to be considered regarding credibility and significance. Afterwards some prominent simulation frameworks are presented that have a 3D visualization and it is talked about their features and drawbacks. The concluding section introduces the MARS platform in detail, which is the targeted application area for this 3D visualization. It presents the unique selling points, use cases and the system architecture of MARS and thereby provides the basics for the following design and implementation chapters.

2.1 Agent-based Simulation

Before we dive into the details of agent-based simulation, it is worthwhile to define what an *agent* actually is. Their working principle is a relatively straightforward one, and probably every reader has already dealt a number of times with agents: Whether it is consulting an employment center, making use of a travel agency, searching online for a flight or booking a hotel – in each of these cases the client asks somebody or some service to act on their behalf.

No matter of the agent being a real-life job broker or a software search agent, the same fundamental rules apply. The agent takes the role to represent its client while performing the assigned task. In order to do this job efficiently and conveniently, an agent is equipped with a certain level of autonomy. It breaks down the task into a set of obtainable goals, observes the current situation and plans its actions accordingly. This plan is constantly revised to take the achieved progress and changing external influences into account. The following cite from Franklin and Graesser (1997) condenses the main characteristics of an agent:

> "An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."
>
> *Stan Franklin & Art Graesser, 1997*

According to Wooldridge (2002) there is no consensus on a uniform definition for the term "(software) agent". However, the following four properties are widely recognized:

1. **Autonomy:** An agent decides on its own, it is not controlled (contrary to an actor).
2. **Independence:** An agent sets and follows its own agenda.
3. **Evolution:** An agent strives to maximize its advantage by adapting its behavior.
4. **Environmentally-related:** An agent is a part of its surroundings and cannot be regarded as isolated, because the environment defines its perception and action options.

In addition to these characteristics expressed in Green et al. (1997), an agent may have further attributes, such as learning algorithms to optimize its planning or mobility features. Another distinguishing criterion is the degree of proactivity and reactivity in its behavior. The standard reference "Artificial Intelligence: A Modern Approach" by Russell and Norvig (2003) differentiates agent behavior into five categories based on the complexity of the reasoning function. This taxonomy ranges from the *simple reflex agent*, which only uses the current percepts and acts according to a set of condition-action rules, over the *model-based reflex agent* (has knowledge about of the world), *goal-* and *utility-based* agents (have explicitly modeled goals, attainment strategies and utilization functions) up to *learning agents* that incorporate a learning element into the agent function. The latter group is the most sophisticated one, able to evolve and improve over time by discovering new solution strategies.

Regardless of the agent's complexity, its main loop is often split up into three phases, called *sense*, *reason* and *act* (SRA). As recapitulated in Gat and Bonnasso (1998), SRA is a long-established partitioning pattern emanating from robot control that provides a clean separation of concerns. Figure 2.1 depicts the agent cycle: First the agent percepts its environment through its attached sensors. Afterwards the reasoning phase takes place, in which the perceptions are evaluated and a response action is determined. This decision logic may be arbitrarily complex and can range from simple decision trees to sophisticated mechanisms like *goal-oriented action planning* (GOAP), as stated above. Finally, the returned action is executed and thus the influence on the agent itself, other agents and/or the environment is applied.

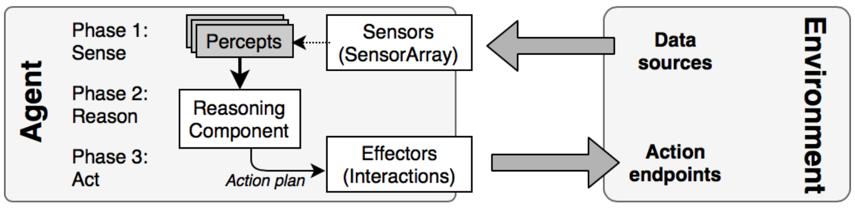


Figure 2.1: Basic sense-reason act (SRA) cycle.

The autonomy and responsiveness of agents entail a number of benefits, which come in handy for the design of complex and adaptive systems. *Agent-based software engineering* (Jennings (2000)) exploits these properties by using hierarchies of autonomous and interacting agents to model such a system. The interplay between the agents achieves various emergent effects, like self-organization or convergence. If the intended purpose is to run a computer simulation, often the related term *agent-based simulation* (ABS) is used.

The core concept of ABS is the realization of the simulation entities as agents, and as such it belongs into the category of micro-simulation techniques. Its heart is the *agent-based model* (ABM), in which the specific behaviors of the simulated individuals are explicitly modeled. Agent-based modeling allows to maintain the structure of the depicted reality in the model and to express pro-active behavior, by which means it found many applications in social and ecological domains (Davidsson (2001)). In contrary to the mathematical models of macro simulations that use averaged input values instead of individuals, the structure of an ABM emergently arises from the agents' interplay. According to Parunak et al. (1998), this makes ABM well-suited for domains dominated by discrete decisions and with a high degree of localization and distribution, whereas equation-based modeling is more suitable for systems in which the dynamics are dominated by physical laws rather than information processing.

While ABM by itself simply implies that agents are used, no statement is made on what an agent portrays: It could be a depiction of a real-world entity as well as a complete (sub-)model or controller class. Most of the time, the *individual-based modeling* (IBM) is used, a refinement characterized by Grimm and Railsback (2013) and predominantly used in ecology domains. The main point of IBM is the focus on the individuals, stating that an agent should only represent a real-world entity, e.g. a human, plant, animal or vehicle. This conceptual similarity makes agent development straightforward for domain experts, because in many fields of ecology and sociology the behavior of individuals has been researched exactly while crowd behavior remains unclear. However, caution has to be exercised during modeling in order to produce desired emergent effects. A set of rules also framed by Grimm (1999) give some guidelines by emphasizing the significance of proper modeling and suggesting a scale-down approach, starting with a coarse model and refining it consecutively by adding details.

With the agents being single individuals, it is essential to create a sufficient amount of agents in total, so that emergent behavior can arise. Many phenomena become only visible at a certain scale and investigations made by Yamamoto et al. (2008) showed that massively increasing the amount of agents can significantly change the outcome of the simulation. Other scenarios are inherently large-scale and can only be done using thousands or millions of individuals, e.g. traffic flow simulation, disease-spread forecasts or predictions for the development of an urban sector. Because of this necessity for many agents, ABS it is also often referred to as *multi-agent simulation* (MAS) or even *massive* (M-)MAS, in case very large simulations are run. Together with the rapid increase of compute power, the ever-growing complexity and the need for forecasts in many domains, multi-agent simulations are becoming more and more popular. In response, a wide choice of MAS platforms emerged over the last decades, from which a few are presented in 2.3.

2.2 Benefits of 3D Result Visualization

“A picture is worth a thousand words” – this famous proverb puts it into a nutshell, how important it is to explain complex issues using graphical methods. In many occasions it is far more easy to express information with imagery than in a comprehensive text, because for humans it is natural and intuitive to perceive information visually (Tory and Moller (2004)). “Seeing is believing” is another one, meaning that the process of observation leads to a *being-there* experience, which leaves a strong impression and a high credibility.

When running complex and large-scale multi agent simulations, the result preparation and presentation is of high importance to make efficient use of the generated data. For these reasons, it seems obvious to make use of visualization techniques for simulation result exploration. Subject of this thesis is the *3D visualization*, which displays the "virtual world" of the simulation model as a three-dimensional scene. This scene comprises of all simulation entities, rendered with appropriate 3D models at their respective positions. As the simulation advances, the behavior of the entities is displayed using computer animation, so it can be actually seen how they move and interact. A free-floating virtual camera provides a view onto the scene and allows the spectator to explore the world and look at the individuals from arbitrary angles (Robertson et al. (1993)). Additional information may be delivered purposefully by clicking at the object of interest, e.g. in form of a tooltip. Together with a timeline or video-player controls (play/pause, forward, back), the user has the ability to freely discover and (re-)play every simulation step and from every perspective.

As it gets evident from the above paragraph, 3D visualization is hardly useful for any other use case apart from individual-based simulation models. For that purpose, however, it fits extremely well, especially for geospatial-related simulations with the agents representing real-world beings (Bijl and Boer (2011)). In these scenarios, there is a natural one-to-one relationship between the real-world entity (let’s say, a pedestrian in an evacuation scenario), the agent representing that being (an instance of the *PedestrianAgent*) and the 3D object depicting it (an instance of a human 3D model, equipped with animations like walking, running, crouching and so on). Because of the indispensable coupling between agents and the environment, also the surroundings have to be visualized, e.g. terrain elevation, roads, buildings and all other parts that are of importance for the agents. These features may be optionally enriched with auxiliary data (e.g. terrain landmarks) or visual effects (lighting, weather) to create a realistic atmosphere that eases orientation and also makes the visualization more appealing to use. All these measures combined can culminate in an intuitive simulation data explorer which makes result evaluation similar to watching an animation movie or playing a *serious game* (Michael and Chen (2005)).

The utilization of a visualization can be categorized into three purposes, as done by Bijl (2009). The first cause is *validation*, which is the process of ensuring that the implemented model is a correct depiction of the targeted domain. According to Balci (1997), visualization belongs to the dynamic validation techniques, which render images of the model's behavior during simulation. With regard to the 3D visualization, this means that the user can get an impression of the ongoing interactions and is able to discover errors by simply observing the scene and looking for discrepancies. Such an option provides a very convenient way to check for movement patterns, agent distributions or if areal boundaries are adhered. Beside the evaluation of the overall validity, a 3D visualization could also be used by the agent modeler during development as some kind of "visual debugger". For example, he or she can perform a simple small-scale simulation run and track a specific agent over time by following its 3D model. The visualization expresses the agent's actions in terms of animation and movement and enables the developer to easily compare the actual (model) behavior with the expected (real-world) behavior.

After the validity of the results is ensured, the *analysis* takes place. Again, a 3D visualization offers a straightforward way to comprehend a model's interdependencies by just investigating what's going on in the scene. The adjustable viewing angle plays an important role for the observation: A higher perspective allows to watch a large crowd from above, seeing patterns and emergent phenomena, whereas a zoom close to earth reveals the individuals and their behavior and properties. At this point it should be noted that a 3D visualization is more of supportive character – it does not make common visual analytics methods superfluous, which display population curves, state distributions and much more over time. The main benefit of 3D visualization is the easy-to-understand presentation form that makes the evaluation of simulation results also accessible to non-scientists.

This directly leads to the third application, *marketing & presentation*. Simulation is done purposefully and in almost all cases, external stakeholders like decision makers or sales managers are involved (Padilla et al. (2014)). These groups often have no scientific background or at least little experience with agent-based simulation techniques and its evaluation. Here, a 3D visualization can prove useful to allow these groups to understand the model. Although they may not be capable of reading out numbers or quoting math formulas, from observing the behavior they get able to predict events and thereby gain confidence in the model, thus promoting its acceptance beyond simulation expert circles. Another large field of application are interactive training simulators, especially used in the military sector or for medical education (Kincaid et al. (2003)). For these domains, 3D visualization plays a very important role because it is essential to have an intuitive interface that allows to grasp the simulation output in real-time and to react to it.

As mentioned before, it is also feasible to add additional elements to the visualization, e.g. 3D models displaying points of interest (reference points for orientation) or texture overlays (like a road network projected unto the terrain). This information integration of simulation results and auxiliary data sources helps to create a close resemblance to the real world and may grant further insights into the model's correlations. Such a high-quality visualization tends to have an impressive and mesmerizing effect, resulting in a valuable presentation tool. However, Banks and Chwif (2011) warned that it is also necessary to exercise caution at this point, because users may be misled by a visualization enriched with additional data – especially if they are not familiar with the simulation domain. In order to create a useful visualization, it is therefore important to attend that the visualization should *facilitate insight* into the data and *provide a knowledge gain* to the viewer. This is achieved by accurately reflecting the simulation model and its input data, marking all supplementary data sources appropriately (Vernon-Bido et al. (2015)).

This chapter gave a concise outline on how 3D visualization works, what benefits it introduces and for which use cases and domains it is applicable. It also mentioned several aspects that have to be kept in mind when building such a visualization. The concrete requirements for this solution are collected in chapter 3.1 and applied in the draft in 3.2. Prior to that, the following section introduces some existing MAS frameworks offering 3D visualization and examines their features and conditions.

2.3 Existing Simulation Frameworks using 3D Visualization

This chapter lists a number of related multi agent simulation platforms and surveys their features and characteristics, with a special regard to their 3D output capabilities. These frameworks are grouped according to availability into free-to-use (open source) software, commercial products for civil use and military solutions.

2.3.1 Open Source Frameworks

One of the most renown open-source simulation tools is NetLogo. Developed by Wilensky (1999), NetLogo is a stand-alone application that combines a model editor and simulation runtime in a common interface. It quickly became a de-facto standard for small agent-based models in the academic sector, especially for student projects and low-budget research groups. Sklar (2007) summarized the features of NetLogo and highlighted its availability (JVM-based, so it runs on any major platform) and its ease-of-use ("simple enough for children to program"). Notwithstanding, NetLogo is capable of building and running sophisticated models, using popular multi-agent modeling languages such as *StarLogo* and a large library of pre-written sample simulations that are shipped with it. In addition, NetLogo is very extensible and allows experienced users to write their own modules in order to add required functionality.

NetLogo has numerous visualization options, among others also 3D capabilities. Kornhauser et al. (2007) reviewed these options, but it becomes clear that NetLogo's presentation options focus on the analytical display and 2D grid-style visualization. Though 3D is possible, it is only rarely used and little support is provided, with the official documentation advising against it.

Next in popularity comes GAMA, the *GIS & Agent-based Modeling Architecture*. GAMA is a Java-written open source modeling and simulation platform for agent-based models in complex environments (Drogoul et al. (2013)). Its strengths are in spatial simulations and many features for the integration of GIS data are provided. Moreover, GAMA has ready-to-use abstractions for the most common needs (e.g. decision architectures and generic behaviors, such as movements) and also offers a dedicated high-level modeling language (GAML) that allows non-programmers to create complex models. The platform is built in a multi-level architecture and, like NetLogo, it can be easily extended by writing custom Java plug-ins.

Compared to other famous open-source and Java-based ABS frameworks like JADE (Bellifemine et al. (2007)) or MASON (Luke et al. (2005)), which have either no or very little 3D presentation options, GAMA supports at least a very basic 3D visualization. Agents can be displayed as simple, textured 3D objects that are rendered on a flat terrain or GIS map. Though it is a tedious task to set up a scene and a lot of coding is involved, GAMA has a 3D tutorial and a comprehensive documentation and outcomes as in figure 2.2 can be achieved.

While the simulation execution of the above frameworks can be considered as "linear", Vigueras et al. (2013) proposes a scalable MAS architecture for interactive applications. The simulation is spread across multiple execution nodes that work in an (almost) autonomous fashion. Each node manages a part of the environment and the agents residing in that area, synchronizing itself with the neighbor nodes only in case of an overarching interaction or transition. The visualization takes advantage of this partitioning and only queries those nodes that are in the current field of view, allowing a potentially massive data reduction and a delivery of near real-time scenes. Few information can be found on the performance of the actual 3D engine, but figure 2.2 suggests that it is of a more basic character.

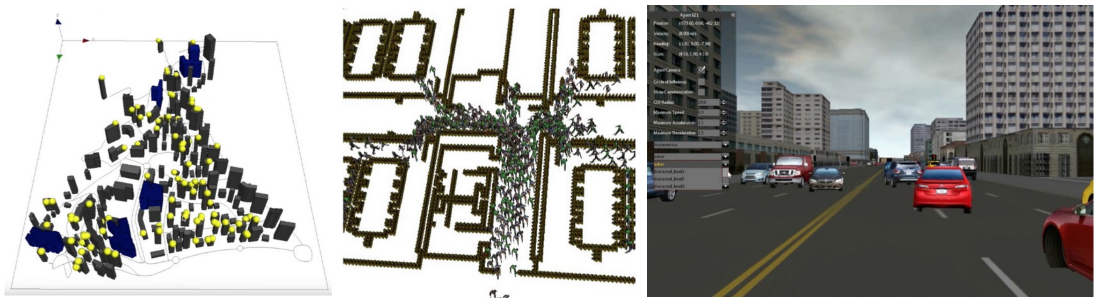


Figure 2.2: Basic 3D capabilities of GAMA (left), a Vigueras evacuation scenario (middle) and the impressive rendering of DIVA/MANTISSE (right). First and last screenshots were taken from the homepages, the middle image originates from the linked paper.

DIVAs 4.0 is a multi-agent based simulation framework developed by the University of Texas at Dallas. It is designed for large-scale ABS in open environments and offers a modular approach to create reusable, extendable components for the control and visualization of simulations (Al-Zinati et al. (2013)). As many other frameworks, DIVAs comes with a set of domain-specific libraries to allow rapid simulation development. The system uses a micro-kernel approach with a pluggable architecture, providing a flexible way to add or remove self-contained modules. DIVAs is fully implemented in Java and uses enterprise-scale technologies (e.g. ActiveMQ and JavaFX) to ensure good scaling properties suited for massive simulations.

Obedient to the modular architecture, DIVAs' 2D and 3D visualizers are separate units, loosely connected to the simulation by means of a message transport service. Both visualizers receive the simulation states through the message bus and generate 2D respectively 3D scenes out of it. Beside displaying the current state, DIVAs also features an interactive editing system, which allows the user to build or modify an environment during simulation run-time.

An application for DIVAs is the MATISSE (Multi-Agent based TraffIc Safety Simulation systEm) project, also led by Al-Zinati and Zalila-Wenkstern (2015). It employs the DIVAs platform to conduct M-MAS simulation models for intelligent transportation systems (ITS). These models are made of a complex traffic network and agents representing the autonomous vehicles, human drivers, intersection controllers and more. Figure 2.2 also contains a screenshot of the result exploration tool, which offers a quite stunning and game-like experience.

2.3.2 Commercial Products

With multi-agent simulations getting more popular in a broader field of application, an increase in modeling and simulation frameworks for civil usage could be noted. Much effort is put into accessing complex domains by employing multi-agent simulation and reasonable visualization – be it in physics simulation, the prediction of disease spread or the optimization of transportation processes. The following frameworks are taken from the 2017 Swain List, a regular series that surveys and compares contemporary simulation software.

AnyLogic is one of the leading simulation platforms for business applications. It is a multi-method modeling and simulation platform that supports *System Dynamics*, event-based and agent-based simulations in any combination. A wide range of industries employ AnyLogic for the simulation of e.g. supply chains, transportation, rail logistics or passenger management. The simulations can be run locally on any major operating system or remotely in a cloud environment that allows high-performance computing and online simulation analytics from any device. In addition, the company behind AnyLogic offers a number of support services, ranging from training sessions to modeling assistance.

To make an easy and fast modeling possible, a wealth of industry-specific libraries exist that can be incorporated. For example, process and material libraries can be used for a supply chain simulation or the traffic and pedestrian libraries to build a road network simulation. It is also possible to integrate GIS maps within the simulation models or to use any major data storage as model input. Furthermore, AnyLogic is fully extensible at Java level and features a concept of reusable models by separating between the model's internal logic and input data.

For visualization purposes, AnyLogic features analytical, 2D and 3D display options. All visualization forms may be integrated seamlessly, as portrayed in the picture series 2.3 and both artificial and real-world environments can be displayed. For the latter it is also possible to use streaming APIs to visualize the terrain with texture overlays (e.g. satellite imagery) and project the simulation entities as 3D objects on top of it.

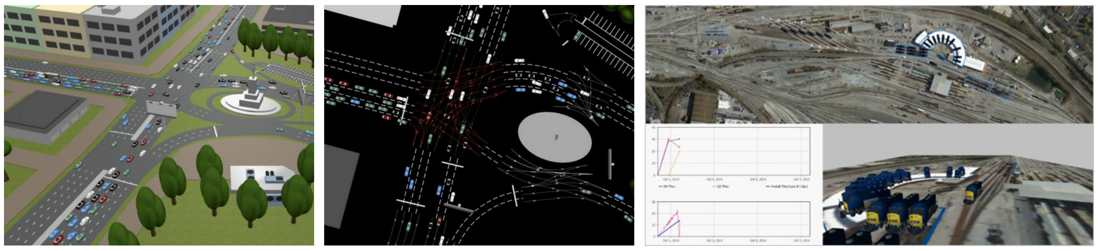


Figure 2.3: Visualizations types in AnyLogic. Left-to-right: 3D, 2D and real-world projections. All images taken from the showcase section of the AnyLogic web page.

Of similar characteristics is FlexSim. According to a product comparison conducted by Capterra Inc. (2018), these two solutions offer an almost identical feature set regarding the simulation capabilities, with FlexSim being restricted to the Windows platform and without cloud execution support. It lacks AnyLogic's extent of data interoperability and import/export options, though it supports the fundamental formats. FlexSim also comes with a large library that contains ready-to-use entities and operations. The software is available as a free trial (limited to 30 objects) and in an enterprise edition.

An advantage of FlexSim is its clean user interface. Many users claim that FlexSim is easier to use than AnyLogic and requires a shorter training period. The model is built in an object-oriented approach and the UI allows drag-and-drop activities to edit an object's properties or to add assets to the model. The logic can be done using pre-built logic building blocks, so that very little or no computer code is required. FlexSim also comes up with a decent 3D visualization that supports more than thirty 3D formats and delivers a superb visual performance. Figure 2.4 shows a screenshot of FlexSim (left side) and one of Simio, which is introduced next.

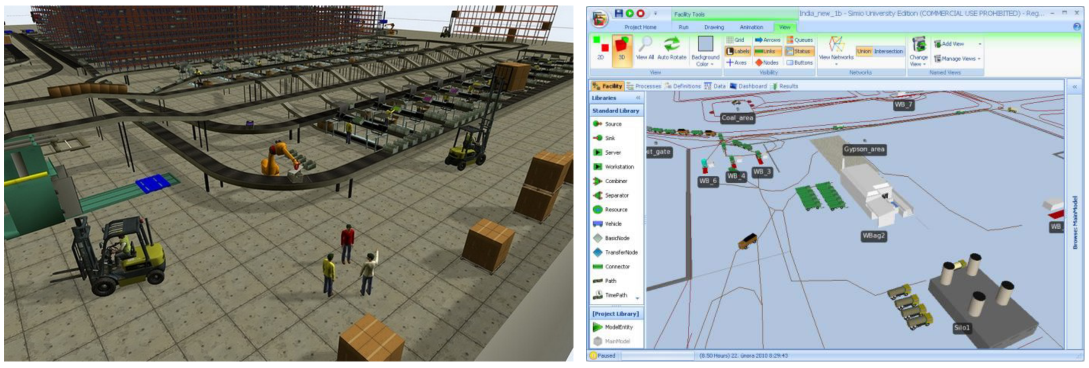


Figure 2.4: Flexsim's 3D visualization (left) and Simio modeling. Images taken from homepages.

The third option presented here is Simio. Like FlexSim, it is also an object-based ABM platform that offers a rich set of pre-built objects and actions with the opportunity to model processes without the need for programming. Simio can be obtained for free for personal and academic use, for the enterprise editions a license fee applies (which is by far the cheapest compared to the two preceding solutions). The company behind Simio operates a SaaS cloud platform called *Simio Portal* that runs on Microsoft Azure and is similar in terms of functionality to AnyLogic's cloud service.

Simio has a large list of well-known companies as customer references, ranging from car and aircraft manufacturers to crude oil companies, healthcare institutions and (air-)ports. On the downside, it does not support the combination of continuous and discrete simulation techniques nor has it the interconnectivity of the above platforms. Regarding the 3D capabilities, Simio lacks behind and few showcases could be found. It has an interface to Google's *3D Warehouse* in order to incorporate models, but the main usage of 3D visualization is for the modeling itself, as shown in figure 2.4 (right screenshot). Still, it is possible to use 3D rendering and animation for the result visualization, though this does not seem to be Simio's primary focus.

Beside these general-purpose platforms, a broad range of specialized MAS software exist. A majority of these packages focus on traffic simulation, e.g. the TransModeler road simulator, the AMATRAK project to envision intelligent autonomous cargo delivery, or CAST, an airport simulation and planning software. Many of these simulation tools also offer a 3D visualization, partly with quite impressive graphics. But because this thesis aims for a universal visualization solution, these specialist systems are not considered here.

2.3.3 Military Solutions

Military and defense organizations have been powerful development drivers in many research branches. This is also true for the computer simulation domain, where endeavors are made since the 80's to harness the computational power for soldier training and to virtually go through different crisis scenarios. Arising from decades of military research in the field of *Distributive Interactive Simulation* (DIS) conducted by the US Department of Defense, a set of protocols and architectures like the *High Level Architecture* (HLA) emerged, which also became an IEEE standard. HLA offers an interface for data transfer between multiple simulation environments (Dahmann et al. (1997)) and supports interactions between the simulation operators and the computer-generated forces. The systems compliant to HLA are primarily designed for combat simulation and early frameworks, such as *ModSAF* (Modular Semi-Automated Forces) by Ceranowicz (1994) had no 3D visualization, but only tactical and strategic displays. 3D engines were still in its infancy and the few add-ons available (e.g. *CommandVU*) had a very limited visual appearance.

One of the first attempts to utilize 3D game engines was the *UTSAF* project by Manojlovich et al. (2003). It is based on the *Unreal Tournament* video game (hence the *UT* in the name), which was (back then) both cheap to afford and easily extensible due to its design to be open for modding. Though the integration into the distributed simulation network by means of a custom DIS connector succeeded, the overall solution suffered of several shortcomings regarding the scalability and rendering performance of that early engine. Since then, computing power and the capabilities of 3D engines have grown rapidly. Two prominent solutions capable of high-realism visuals are presented on the next page.

VT MÄK is a leading company for modeling and simulation software in the military sector. They develop a broad range of products for distributed simulations, which can be flexibly integrated with each other to meet the customer's needs. Beside the products themselves, VT MÄK offers training and simulation building services. Their solutions are used by the US Armed Forces, NASA, aviation industries and armament manufacturers.

The heart of the VT MÄK product suite is *VR Engage*, a multi-role virtual simulator that runs on top of commercial-off-the-shelf (COTS) hardware and is compliant to the IEEE HLA and DIS standards. Around the simulator exists a number of add-ons, ranging from physically accurate sensor and radar models to data integration plugins, modeling tools for characters and environments and analysis clients. The 3D visualization is of game-like quality with detailed human characters, realistic animations and remarkable visual effects, such as changing weather, waves and dynamic lighting. See the first two images in figure 2.5 for a few impressions.

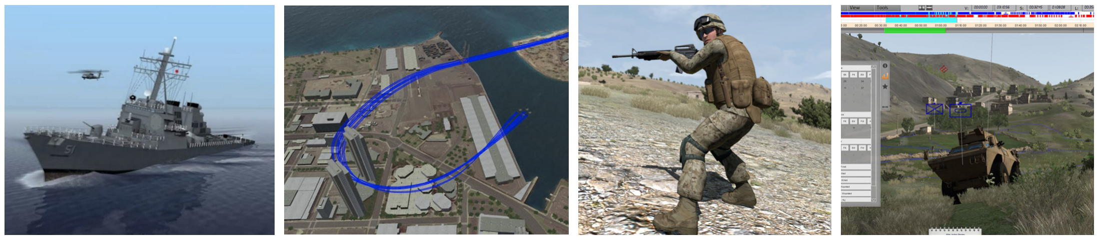


Figure 2.5: Impressions of the visual quality in VT MÄK (left side) and Virtual Battlespace 3. Images taken from their respective web sites, as linked in the text.

Another widespread solution is Virtual Battlespace, developed by Bohemia Interactive Simulations. Currently available in its third version, VBS3 is a tactical trainer that is used by major military organizations, such as the U.S. Army, U.S. Marine Corps and the UK Ministry of Defence. It features a desktop-based simulator that runs on commodity hardware and unifies tactical training, experimentation and mission rehearsal in a comprehensive package. If desired, the functional range can be extended with a multitude of additional plugins, e.g. artificial intelligence toolkits or interfaces for virtual reality hardware.

The core of VBS3 is a distributed multi-agent simulation system that offers interoperability with external HLA/DIS networks. It can simulate any environment for military training and supports massive terrains with large numbers of entities. VBS3 comes with level and scenario editors to create custom missions and provides a huge 3D model library with over 10,000 detailed and animated models, making it a virtual sandbox. The 3D rendering is based on a self-developed engine (which is also used in the tactical shooter game *ARMA 3*) and offers a highly immersive user experience, as also showcased in figure 2.5.

2.4 The MARS Simulation Platform

The output and visualization concept presented in this thesis is intended to fit into the MARS system. A brief overview on MARS was already given in the introduction. This section now deepens this knowledge by delivering more details on the platform, its unique selling points, the typical user workflow and the internal structure. Many of these characteristics are of importance for the design and implementation decisions explained in the following chapters.

2.4.1 Why You Should Use MARS – Characteristics and Use Cases

MARS is a highly-performant platform for multi-agent simulations, developed by the MARS Group at the Hamburg University of Applied Sciences. It comprises of the simulation engine itself (codenamed *LIFE*, developed by Hüning (2016)), a set of optional components and a service and execution infrastructure, unified under a common interface.

The front end to the MARS platform is a web-based interface, offering a sleek and functional user experience. Because large and complex simulations are usually developed in teams, this web suite is multi-tenant capable and supports collaborative work with project sharing options, visibility settings and user role policies. Apart from the modeling itself (see 2.4.2), all steps can be accomplished in the web browser, so the installation of separate tooling is not necessary. This means the simulation preparation, execution and evaluation can be done from every device and all over the world, as long as an internet connection is available. Also the compute power directly available to the user is of no importance, because the simulations are run in the cloud.

This makes MARS especially well-suited for large-scale simulations. The simulation run is executed in a Kubernetes cluster and supports a number of distribution strategies, making it very easy to scale both horizontally and vertically. Big-data features like complex event stream processing (CEP) and distributed storage ensure that the result flood is kept under control. All this complexity is abstracted away from the user with the WebUI being the single point of access to simulation run administration and result evaluation. The analysis tools currently available comprise of a visual analytics dashboard, a 2D map projection, a python interpreter and plain CSV file download for offline evaluation. The visualization presented in this thesis shall extend this portfolio with 3D capabilities.

As an example use-case for both the distributed cooperation and the large-scale requirements serves the interdisciplinary research project ARS AfricaE. Started in 2015, this joint project of European, African and US-American research institutes aims to model South African savannah ecosystems in order to investigate their resilience to external interferences and employs MARS to run massive multi-agent simulations with more than 5 million agents. Another very recent project initiative named Smart Open Hamburg targets to model urban traffic flows, making use of MARS' numerous GIS integrations, which are detailed later.

In addition to these technical features, MARS also offers an intuitive modeling paradigm. The core of that concept is the separation of the model's ingredients into *agents* and *layers*. Every type of self-acting entity (in other words: *individual*, refer to 2.1) is expressed as an *agent* type which gets instantiated and executed during runtime. Agents are grouped onto *layers*, which partition the model into its distinct aspects. A layer can be thought of an information overlay, similar to those known from map services like *GoogleMaps*, and contains either one or more agent types or environmental data. This separation of concerns is a proven methodology from the software engineering and eases the model conception and implementation as well as its future extensibility.

Figure 2.6 illustrates this partitioning for a model used in the ARS AfricaE research project. It features elephants, trees and water holes realized as agents, using terrain elevation and various time series data. Every agent type resides on its own layer, as well as the data sources.

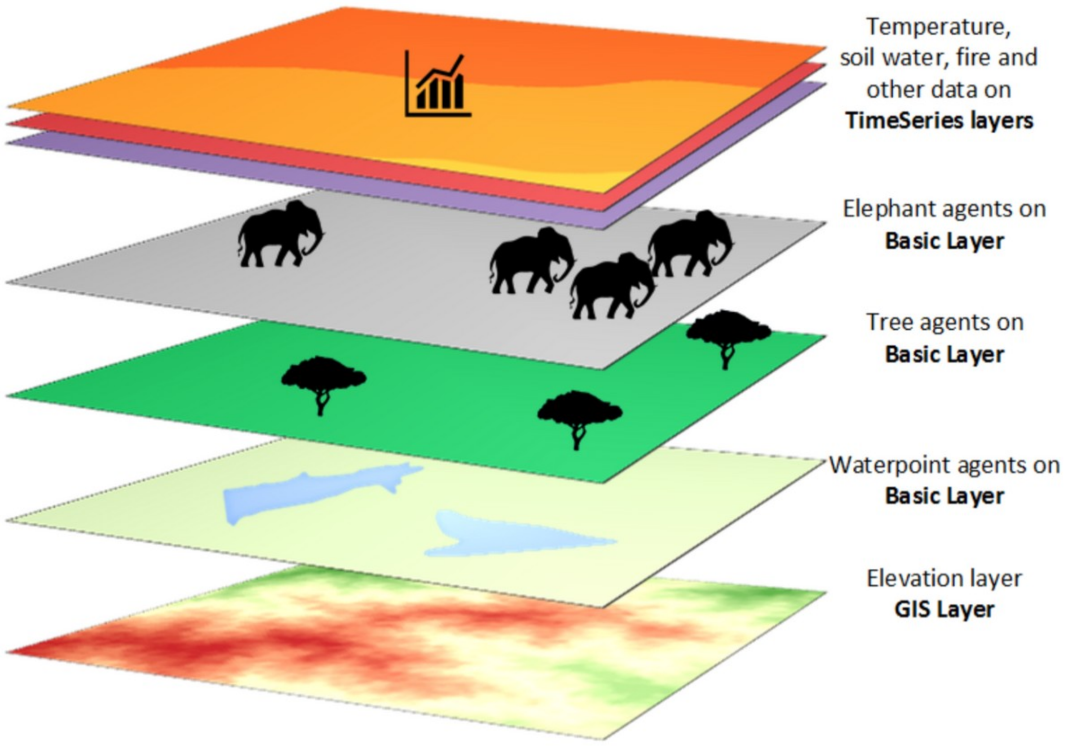


Figure 2.6: The MARS layer concept, exemplarily shown for the *ARS AfricaE* savannah model. The image was taken from Hüning et al. (2016).

To speed up the model development, MARS offers a set of base types that provide core functionality and can serve as a basis for custom implementations. For the agents, the author developed a *BasicAgents* package, which incorporates the SRA agent cycle and is detailed in Dalski (2017b). Advanced proactive behavior can be achieved by integrating using goal-oriented action planning (GOAP), elaborated by Niemeyer (2016)). Such logic building blocks come in handy to model human behavior, e.g. for pedestrian dynamics and evacuation scenarios.

For the majority of the mentioned domains, it is essential to handle geospatial data. The agents are located at real-world positions that need to be represented within the simulation framework. The *BasicAgents* package named above provides support for GPS placement and movement modules, featuring coordinate calculation and conversions. In addition, MARS comes with CPU and GPU based environment representations, providing position management and optional collision detection features with 2D grid, 3D cartesian and GPS capabilities.

Almost all but the most basic simulations rely on external time-series data. To make these measurements or prognoses available to the agents, MARS allows to import CSV files into time-series layers, expressing e.g. climate data such as temperature or precipitation. Beside the temporal resolution, many time-series data also have a spatial reference, specifying the place and the time period they are valid for. In order to access a time-series value, two transformations are required: The corresponding real-world time has to be calculated from the simulation tick counter and a rasterization of the querying agent's (continuous) GPS position to a (discrete) grid cell is needed, taking the resolutions of the time-series for both dimensions into account. Figure 2.7 depicts this rasterization for the spatial dimension, leaving the time aspect out of consideration.

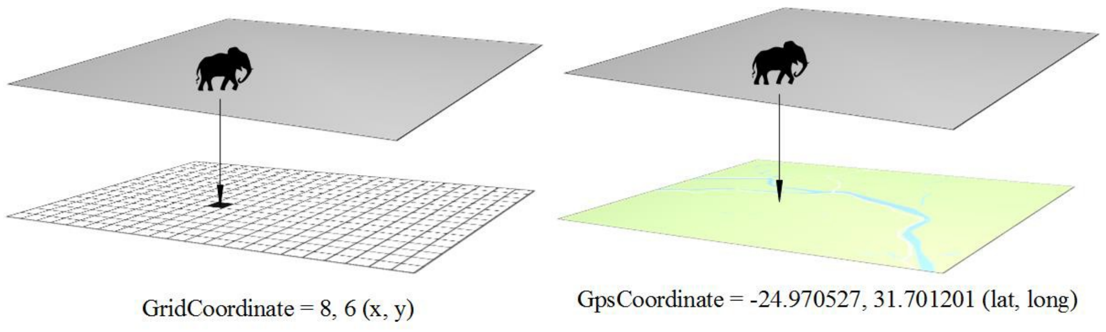

GridCoordinate = 8, 6 (x, y)
GpsCoordinate = -24.970527, 31.701201 (lat, long)

Figure 2.7: Grid & GPS support to express real-world data and spatial rasterization

MARS provides a simple time-series layer as well as performant GIS vector and raster layers developed by Karsten (2018), which can be found in the optional *Components* package. All of these layers are configurable and can be initialized from CSV respectively SHP and ASC files.

2.4.2 Typical Modeling & Simulation Workflow

The work with MARS can be split up into several stages consisting of model development, parametrization, simulation, analysis and error correction or parameter fine tuning, until the desired outcome is achieved. Drawing 2.8 shows an overview of the phases and cycles in a usual modeling and simulation workflow.

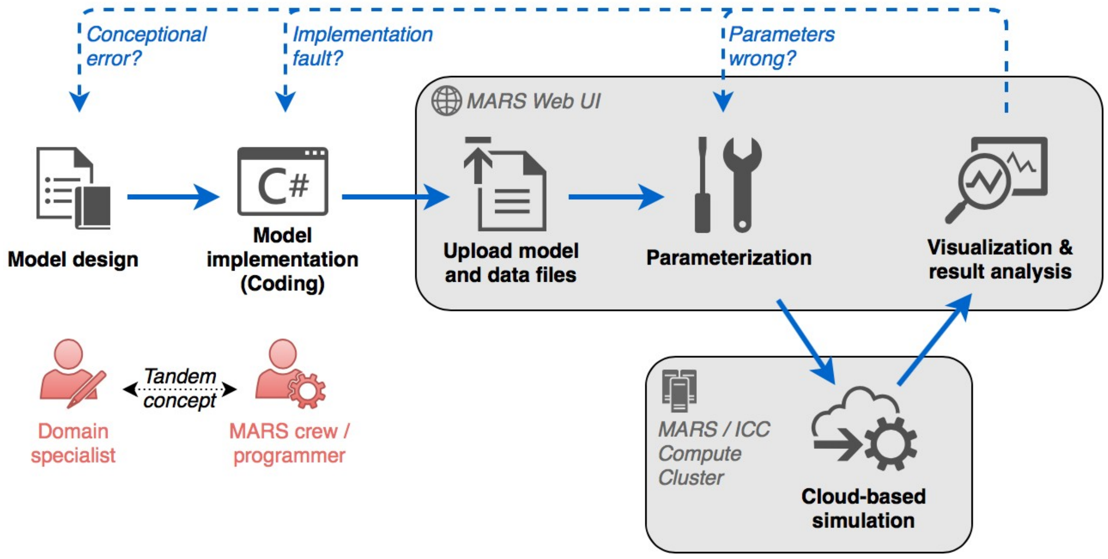


Figure 2.8: The MARS MSaaS workflow

The workflow starts with the model: At the beginning, the model design and conceptional draft is done, splitting the different aspects into layer and agent types according to the MARS modeling paradigm. Though several "modeling as a service" approaches are in discussion, currently the only way to build a simulation model is by writing program code on the modeler's local machine. This can occur either in plain C# or in the MARS DSL (domain-specific language) developed by Glake (2018). At the current stage of the MARS project, there is no graphical model editor available, but such a modeling frontend is feasible and will likely come up in the future. Until then, the previously mentioned tandem concept offers a makeshift for those unfamiliar with writing code on their own.

After the model (or a first draft of it) is executable (syntax checks OK), it has to be compiled, compressed and uploaded to the MARS cloud. For this and all following steps, the user needs an account for the MARS platform and has to be logged in to the web interface available at https://mars.haw-hamburg.de. The platform is organized in *projects* and holds all models, related data, configurations and results.

Beside the model code, also all files referenced in the model (data sources for layers or initialization files for agents) need to be imported. With all required files present, it is time to bind them to the respective simulation entities: This data-to-variable mapping, hereinafter called *scenario definition*, is done by performing straightforward drag-and-drop operations (data file $\rightarrow$ layer; CSV column $\rightarrow$ agent constructor parameter). Also the simulation interval comprising of start and end date as well as the tick resolution have to be set here, specifying the number of ticks to execute and the query range for time-series layers.

Based on the scenario definition, one or more *simulation runs* can be instantiated. A simulation run is a distributed execution in the MARS computing cluster (see also 2.4.3). The simulation results are saved in a database, the log messages (console output) are directly fed back to the web UI and can be read and tracked. As soon as the first results become available, the user may browse to the evaluation tools and analyze the simulation outcome. Screenshot 2.9 below gives an impression on the corresponding UI dialogues in this workflow.

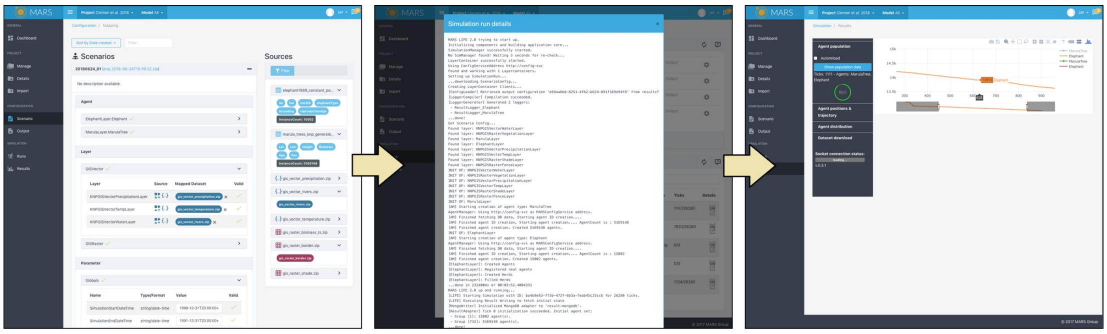


Figure 2.9: Workflow in UI: (1) parameter mapping, (2) execution, (3) evaluation

Building a simulation model is an incremental process and it is unlikely that the simulation outcome is as expected in the very first run. Probably some abnormalities or strange behavior can be observed, resulting from some sort of error: The model code may have some logical flaws or careless mistakes or the conceptual model could be based on misguided assumptions. Also the input files might be faulty or some parameter adjustments are necessary. It is the task of the model developer [and tandem partner] to localize the cause of the error and handle it accordingly.

As soon as the error has been identified and fixed, the cycle repeats until the results are trustworthy and satisfying. It is the hope of the author that this process might be eased and speeded-up with a 3D visualization as exploration tool.

2.4.3 System Architecture

The overview in section 2.4.1 already mentioned that MARS consists of two major parts, the cloud platform for setup, execution & evaluation and the simulation framework called *LIFE*. Figure 2.10 gives a more in-depth insight into the structure.

Regardless whether the DSL or plain code is preferred, the modeling project itself is an executable C# program for the .NET Core platform. It contains the model code and several binaries of the *LIFE* package. Mandatory are the *LIFE Core*, which is the actual simulation engine with the execution logic and the *LIFE API*, providing the interfaces necessary for the generic core to access the model. *LIFE Components* contains the pre-built assets and is optional. All three parts are distributed through the *NuGet* package manager and can be easily installed. The compilation is compressed and then uploaded to the cloud platform.

The *MARS Cloud* is composed of a number of services and databases, which run as Docker containers inside a Kubernetes cluster operated by the HAW. Following the *microservice approach* (detailed in chapter 4), every service has its own task, e.g. data import, user management or simulation run control. An Angular 4 application serves as gateway and encapsulates the access to all functionality in an integrated web interface. The simulations themselves also run as Kubernetes pods in the cloud, connected to the various input databases, the output target and the supporting services. Internally, the run container includes the .NET Core runtime as well as the uncompressed simulation code previously uploaded.

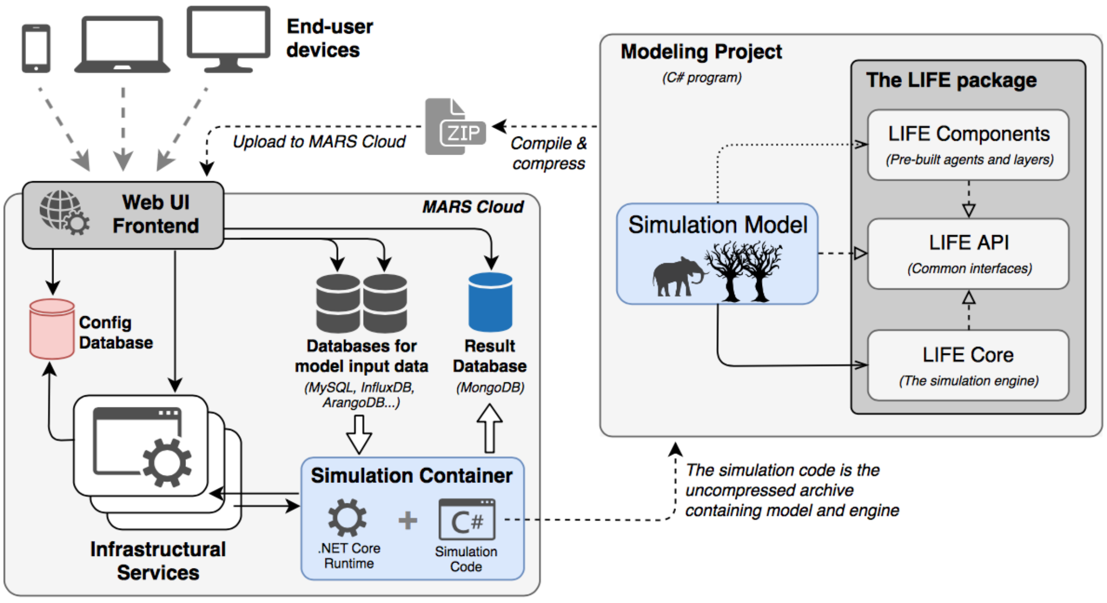


Figure 2.10: Overview on the MARS Cloud and LIFE package.

3 Design

Now that the groundwork has been laid, this chapter presents the draft for the MARS 3D visualization system. First, it collects the requirements that the solution has to fulfill and how the integration into the remaining MARS ecosystem should happen. On that basis, the system gets partitioned into several subcomponents which are outlined afterwards.

3.1 Requirements

This section states all the demands put on the 3D visualization and the underlying output system. They are divided into *functional* requirements, specifying what range of functions is expected from the solution, and *non-functional* requirements or *system qualities*, expressing constraints and how the solution should perform. The subsequent lists name the requirements and explain what they mean and why they are considered important.

3.1.1 Functional Requirements

- **User-controlled output & visualization:** Because of its generic nature, MARS does not have any information on the concrete simulation model, so a configuration system for the simulation output and the 3D visualization is needed. In this dialogue, all essential aspects of the agents and the environment have to be presentable, while allowing the user to pick those relevant to him or her. The simulation engine has to be extended with a dynamic output module that adheres this selection and only persists the requested data. In addition, the 3D visualization needs to be configured, e.g. the 3D models or camera properties have to be set. However, the visualization system shall be optional and pluggable – if not desired, it should provide no additional overhead to the user.

- **Agent visualization:** With agents being the core entities in multi-agent simulations, their visualization is undoubtedly the main focus. An agent is portrayed using a previously defined 3D model, rendered at the position and orientation the agent has at that moment. It should be possible to select an agent in order to highlight it. When highlighted, the visualization shall allow to inspect the agent's custom attributes and follow that agent if the scene advances. An agent grouping according to their home layer seems useful, for this allows a layer-based toggle whether a specific agent type shall be displayed or not.

- **Terrain visualization:** In order to display a realistic visualization scene, also the terrain has to be rendered. In its basics, the terrain is a mesh representing the elevation with a texture laid on top of it. Regarding the elevation and texture, it should be possible to either specify custom data (GIS file or image) or to tap open data APIs to stream real-world scenery. In the latter case, the scene is augmented with auxiliary data not used by the simulation, which has to be marked accordingly to prevent user confusion.
- **Layer visualization:** Apart from agents and terrain, it is deemed useful to display the property of time-series or GIS raster layers as well. They represent information sources like temperature or precipitation, which may be visualized using coloring algorithms.
- **Geospatial (GPS) support:** MARS' strengths are the simulation of geospatial real-world scenarios, which makes GPS support mandatory. Because 3D engines operate in a cartesian coordinate system, an internal abstraction and conversion algorithms have to be developed. In this context a high level of accuracy is crucial, because even very small coordinate deviations may result in several meters of incorrect positioning.
- **Cartesian (3D / 2.5D) support:** Apart from real-world scenarios, MARS also allows arbitrary cartesian environments. Regardless whether it is a continuous (X,Y,Z) space, or just a 2D grid-based system – since MARS can simulate these environments, the visualization should handle it as well. 2D scenarios might be rendered using 3D objects on a flat plane (also called "2.5D"). Similar to above case, conversions from the simulation coordinate space into the engine space and vice versa have to be defined.
- **Browser-based access:** The entire MARS platform is accessed by means of its web interface, so it must be possible to also run the 3D visualization from within the browser to allow a proper integration. No installation of additional plug-ins (Adobe Flash, web players etc.) shall be necessary and the visualization shall run in every major browser in order to avoid any restrictions regarding the accessibility.
- **Simulation live tracking and play-back:** The visualization shall serve as an evaluation tool for both past and present simulations. Saved simulation runs are presented to the user and can be loaded from a database, offering playback and seeking options for all ticks. Equally, the connection to a live-running simulation shall be possible, visualizing ticks in real-time as soon as they have been simulated.

- **Unrestricted simulation exploration:** One of the unique features a 3D visualization offers is the ability to explore every step of the simulation from arbitrary angles and positions. The user shall be able to freely move around, inspecting the scene from any point and zoom level while a set of tick selection controls allows to choose which simulation step is displayed.

3.1.2 Non-functional (Quality) Requirements

- **Maintenance:** This point comprises all practices of good software engineering: The output subsystem is a central part of the simulation system and the visualization hopefully will be used beyond this thesis, so it should be easy to modify and extend the software. Further development, such as adding features or replacing parts (e.g. swapping out the 3D engine with a better one) can be archived by a profound architecture using a modular approach with well-defined responsibilities and a loose coupling.

- **Extensibility:** Whereas above point relates to modifications of the software itself, this item is about content: It should be possible to add or modify 3D models and others assets (e.g. skyboxes) without requiring any program changes.

- **Scalability & efficiency:** MARS is designed for large-scale simulations and hypothesis H2 (section 1.4.1) claims that this should pose no problem for a 3D visualization. This makes it crucial to apply data reduction and caching algorithms on the visualization side as well as a performant rendering engine. For the output system, an efficient storage format has to be found and streaming techniques for the transmission plane are needed.

- **Usability:** As stated in section 2.2, a 3D visualization shall allow a straight-forward result evaluation, also suited for non-scientists. This means the visualization should be easily operable with no learning in advance necessary. Therefore the focus is set on sleek and comprehensible user interfaces limited to the essential data and an intuitive 3D space exploration. The controls and their function should be self-explanatory and the tick selection and movement in the 3D space shall be inspired by video and game controls, creating a familiar setting for many users.

- **Visual performance (*Look & Feel*):** Beside the ease of use, the 3D visualization should also be an enjoyable experience. This shall be accomplished by providing an appealing and convenient user interface as well as a decent 3D quality. Beside the visual appearance, a low response time is desired to allow a smooth exploration.

3.2 Outline of the Visualization Pipeline

This section presents the pipeline that was devised to equip the MARS system with 3D visualization capabilities. It aims to provide a simple, yet effective solution while meeting the requirements listed aforesaid. For that, several strategies were developed, which are detailed in the succeeding section 3.3. But because it is easier to grasp the concepts if there is an understanding of the overall layout, the overview on the pipeline is given first. It consists of a five-tiered approach as shown in figure 3.1 and explained subsequently.

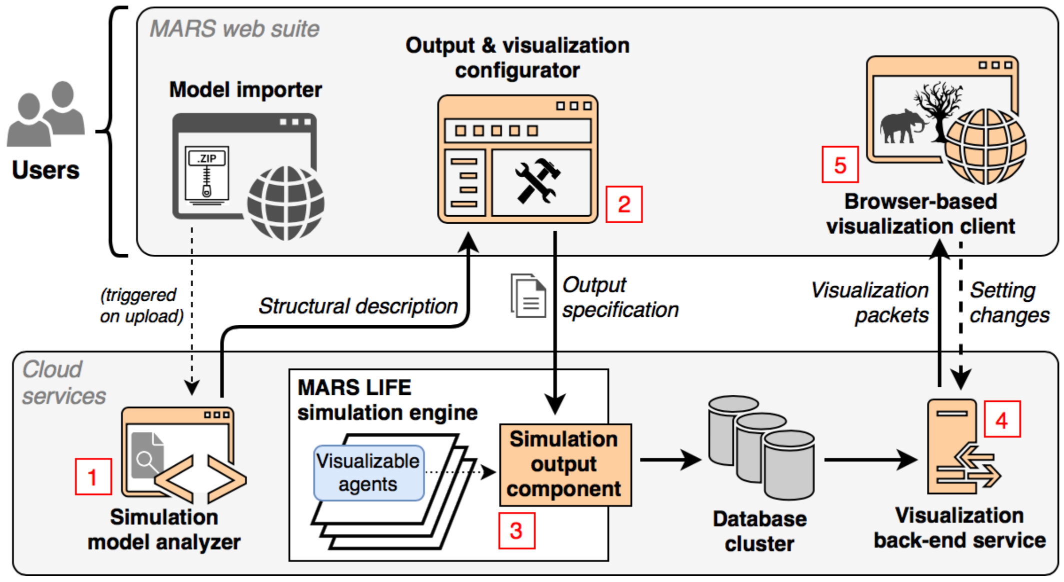


Figure 3.1: The output and visualization pipeline in overview

Because the modeling phase currently occurs offline (see section 2.4.2), the MARS platform has no knowledge on the user's simulation model and its attributes. This makes it hard to offer any purposeful output or visualization, for the simulation has no opportunity to access the model-specific agents and their properties. That is why the entire process starts with a **simulation model analyzer (1)**. As a new addition to the MARS cloud infrastructure services, this tool has the task to perform a model code scan and generates an exhaustive description on all simulation entities and their properties. Invoked automatically on model import, these information are stored alongside with the model in the database and are made accessible to all further processing.

The next part of the visualization pipeline is the **output & visualization configurator (2)**. It plays a significant role in resolving the large-scale issues mentioned before by incorporating several storage saving strategies, which are detailed in 3.3.1. In a nutshell, the basic idea is to offer the users a choice of what to output and how often that should occur. For that purpose, the configurator reads the model description generated by (1) and generates an output configuration dialogue, presenting the layers, agents and their attributes in a selectable fashion. Integrated into this view is also the visualization setup, which is split up into a general section for the common settings (camera, terrain, movement speed) and in type-specific options that are presented alongside with the output config, such as the 3D model to choose or its scale. Because of their resemblance and dependence (you can only visualize an agent if you have chosen to output it), it was considered appropriate to put these two in itself distinct setups into a joint interface that gets part of the MARS WebUI.

As soon as everything is set up, the simulation run can start. On simulation bootstrapping, the **simulation output component (3)** acquires the output configuration and prepares itself to be able output the mentioned types. At that point a technical problem arises, because it is not feasible to access a concrete agent attribute (e.g. the biomass value of a tree) from within a generic runtime. A solution is sketched in 3.3.2, it rests upon dynamic code generation and self-expansion in order to gain the required output capabilities. Once initialized, the output component loops after each simulation step over all entities and writes their results into the database, making them available for visualization.

The 3D visualization system consists of two parts, a server and a client component. The **visualization back-end service (4)** runs in the MARS infrastructure and forms the link between the visualization clients (the users) and the database containing the simulation results. Its responsibility is to acquire and prepare the requested data and to provision it to the clients over the network. For that purpose it offers an WebSocket endpoint, facilitating information queries and data retrieval via a full-duplex connection. Because a wide range of query, control and supply messages have to be sent, a lightweight and efficient information interchange protocol was designed, which is detailed in 3.3.4.

Finally to the actual 3D visualization: The frontend is a **browser-based client (5)** that contains the 3D rendering engine and the user interface to control the visualization. When a user starts the 3D visualization client by opening its web page, it automatically establishes a connection to the backend service and queries the available simulations. Heeding the requirements of section 3.1, the visualization allows the user to freely explore the simulation output in a continuous, three-dimensional space by showing all the agents and environmental features like terrain and obstacles.

Because a large number of entities is expected, the visualization has to incorporate load-reducing techniques on both sides. Section 3.3.3 presents several strategies for the rendering and the asset management to handle the big-data issues.

3.3 Concepts

Building on the collected requirements and preceding component draft, this section presents several ideas to meet the demands put on the visualization pipeline. It begins with a storage-efficient result output approach, proceeds to the components accessing and visualizing these data and also details the network protocol used for the server-client communication.

3.3.1 Storage Saving Strategies

The savannah ecosystem model used in section 2.4.1 for the explanation of the layer methodology is of a very large scale and contains over 5.5 millions of agents in total. A complete output by simply dumping the entire attribute space of every individual to the database would result in more than 700MB of data to be written in each tick. Beside the waste of storage space, also the simulation process is slowed significantly and the evaluation tools have to process this vast output, making them slow as well. Especially for a real-time capable and browser-based (thus streamed) 3D visualization, this naive approach is not feasible and a more advanced solution is urgently needed.

This section proposes three major improvements in order to reduce the data volume. First of all, instead of always saving everything, the user has to explicitly enable the output for an agent or layer type. In this selection, they also specify the output frequency (if a more coarse resolution suffices, there is no need to save in every tick) and the actual data to output, like the spatial attributes (if available) and the output-eligible properties. The latter are confined by convention to be publically readable, primitive attributes (i.e. plain values, no complex structures) to make them programmatically determinable. This process allows to save storage, bandwidth and runtime by limiting the data to those relevant for the evaluation and is integrated as a new configuration dialogue into the MARS web-suite. Refer to the overview in section 3.2 or to section 4.2.1 for implementation details and screenshots.

A second strategy focusses on the change behavior of the attributes and distinguishes between *static* (also called "fixed") and *dynamic* output properties. Static properties are those of immutable character; like meta information on the agent (e.g. type, GUID, layer reference) and all user-defined attributes that are only set once during initialization – e.g. the herd ID of an animal read from a CSV file, a randomly chosen sex or the GPS position of a tree loaded from a spreadsheet. Whereas the first two examples refer to custom agent attributes, the latter is a special case. The differentiation between moving and stationary entities is not only relevant for an optimized storage, but also for the 3D visualization: An immobile object will always keep its initial position and orientation (trees normally won't move) and thus can be rendered much more efficiently by keeping them in the scene hierarchy between frame transitions.

Though this static/dynamic differentiation allows to massively reduce redundancy, it also has two drawbacks. First, this process has to be done manually and therefore results in additional user interactions by marking the static attributes in the result configuration dialogue. Second, it introduces a possible error source resulting from incorrect configurations. If the user fails to mark a fixed attribute accordingly, it results in storage waste with this property being written unnecessarily often (old output behavior). On the other hand, if a dynamic property is marked as static, it is only written once and ignored for the rest of the simulation. This case is especially dangerous, because it results in missing data and the user may implicitly assume that the results are correct and just no change occurred. A feasible alternative detailed in section 7.3 would be an improved model code analysis, which traces write operations for each attribute throughout the entire source code and automatically determines the right setting.

Building upon the differentiation between static and dynamic properties, it can even be expected that not all dynamic properties change in every tick. With this in mind, a third strategy introduces a refinement that is based on full state and delta state outputs, a technique borrowed from the realm of video formats. It uses *key frames* (I-frame in the video domain) to save full-state expressions of an entity in regular intervals and *delta frames* (P-frames), which are sparse structures and only contain the changes in respect to the previous frame (Le Gall (1991)). This technique comes in handy to further reduce the amount of result data by focusing on the state changes between the simulation ticks, but also adds to complexity: In order to find out what has changed, some sort of comparer is needed that tracks all variables. Every time a variable changes, the new value has to be recorded in the delta frame and adopted for the next comparison. The changes are saved as absolute values, thereby eliminating the needs for offset calculation at output and value summation at retrieval.

In order to facilitate a quick search on the results, the key frames have to be written in regular intervals (every n-th tick). The key frame to delta frame ratio is an additional configuration parameter with significant impact on the amount of data and the processing time needed for retrieval. These two goals are conflicting – a high delta proportion helps to reduce redundancy, but craves for an extensive and thus computationally expensive aggregation. In Dalski (2017a), several experiments were conducted, showing that a high ratio is better suited for simulations with few state changes per time step or only a small amount of entities involved, leading to storage savings of up to 66%. Models with lots of interactions, however, have less potential for savings (30-40%) and frequent key frames seem favorable in terms of data access.

Details on the data output and retrieval using key- and delta frames are given alongside with an example in the implementation chapter in section 4.2.2.

3.3.2 Dynamic Output Module Generation

Regarding the output of the simulation results, basically two ways exist. You could either build a component that is part of the simulation system and iterates over all entities or you could shift the responsibility to the entity itself. In the first case a generic output solution is needed, because at design time of the simulation system no information on the user-written agents and their properties is available. The other way, however, would mean that the user has to take care of the output by either implementing some result interface for every type or by accessing the database directly. This is not only cumbersome and error-prone, but also contradicts the idea of providing a domain-independent and convenient simulation platform.

For these reasons, only the first alternative is considerable for MARS. It raises a problem, though: How is it possible to access a concrete agent attribute (e.g. the biomass value of a tree agent) from a generic runtime? Furthermore, in order to implement the storage saving strategies from 3.3.1 some logic exploiting the fact that an attribute is either static or dynamic is needed, as well as a comparer to detect changes in an attribute's value. As a consequence, there is no other way but to build specific output modules for each agent type that are tied to a concrete instance during runtime.

The solution to obtain these type-specific output modules (in the following called "logger") is to create them on-demand during the simulation system initialization. With the model structure analyzed on import and the output configuration done by the user (components 1 and 2 in figure 3.1 of the outline), the system has all the required information on the agent types, properties and change behavior to its avail. On startup, these information are downloaded and then used to write logger code for every type. This is done by using a default logger template and filling its method stubs with the particular details. The purpose of the template is to provide a class frame that complies to a common logger interface used by the generic part of the output component. In this interface, three output returning functions are demanded, respectively addressing the agent's metadata (containing also the static properties), key frames and delta frames.

After the logger definitions are done for all types, the sources are compiled by using a *compiler-as-a-service* technology that generates a binary containing the logger code. This binary is then re-injected into the runtime context and its contents are made available through a technique of source code reflection, similar to that one used in the model code analyzer. After this process, the output component is aware of the concrete loggers, retrieves their executable modules from the assembly and stores them in an $agenttype \rightarrow loggertype$ mapping.

Equipped with the logger modules, the output system is now fully initialized and ready for the registrations of the simulation entities. On registration, the entity's type is determined and the corresponding logger module fetched from the mapping. A new instance of this module is created and tied to the simulation entity by passing its reference along. The output component maintains a list of all instantiated loggers and iterates over them to gather all results. More about the output process can be found in the implementation chapter, section 4.2.1.

3.3.3 Visualization Performance Improvements

The requirements analysis in section 3.1 made clear that the pipeline has to handle large-scale scenarios as well as multiple data sources. After the previous solutions aimed for a reduction of the output volume, this part treats the visualization components.

As already sketched in the outline in 3.2, the visualization is made up of a browser-based client and a backend service. The server-side part does the preprocessing and reduces the amount of acquired and transmitted data by restricting them to those perceptible by the user. This *interest management* uses the client's camera position plus an additional buffer zone for spatial queries of agents and terrain. To allow such a filtering, a client management has to be implemented that keeps track of the clients' positions, settings and selections, as well as a record on the data already sent. Whenever changes occur, the visualization client notifies the server and receives the new payload in turn. All this communication is done using the network protocol specified in 3.3.4.

In order to reduce the network overhead, batch-loading is used for data transfer. The agents and terrain data to transmit are collected and transmitted as chunks. For this purpose, the server build various zones around the camera position. Beside the already discussed field-of-view and buffer zone a third zone is introduced, which is bigger and contains all objects that are pre-selected for the next transmission. An even larger fourth zone defines the caching area: All entities in it are kept in memory on client side to allow an immediate re-rendering, if necessary. This helps to reduce the retransmission of previously sent data in case the user returns to an area recently visited before, e.g. on zoom operations.

On the server side, caching is mainly for the terrain streaming of importance. The usage of external data sources introduces an additional bottleneck, because the data queried by the client have to be fetched via internet first before they can be delivered. Since almost all simulations take place in a defined area, a terrain visualization without caching would cause recurring queries of always the same elevation and texture tiles. A local availability helps to circumvent these time-consuming downloads and allows a high degree of independence from external sources. To keep the server's disk space in check, a configurable cache manager is used that employs two deletion policies, LU (least used) and LRU (least-recently-used - longest idle time since last access).

Whereas all previous techniques targeted to accomplish an efficient preliminary selection, also the rendering can be improved. To further confine the area to render, the visualization client performs *view frustum culling* (Assarsson and Möller (2000)), which reduces the number of entities to those actually visible. It uses the camera view cone + viewing range and also skips all objects concealed by others. Only the agents, objects and terrain tiles which effectively appear on the screen remain in this selection. It still would be a waste of performance to draw all these models with the same *level of detail (LOD)*, because models far away only take in few pixels of the image and a lot of details would diminish. As a consequence, a simplified model with a reduced the polygon count (lower LOD level) would be much faster to draw without reducing visual quality. Further optimization can be achieved with *instancing*, which performs an aggregation based on 3D model and LOD to allow a batch processing in the engine's internal main loop. A once loaded 3D model can be reused for all entities of the same kind, resulting in less GPU buffer uploads and drawing calls.

3.3.4 Network Protocol

For the communication between visualization client and server it is necessary to allow an independent data transfer in both directions: User inputs like simulation- or tick selection are relayed to the server as well as position updates or agent selections. The server replies with data packets, which can be sent asynchronously or as direct response. But it must also be able to push data to the client (on automatic play-back) and to send update messages on simulation availability and progress. This would not have been possible using conventional HTTP connections since these are client-initiated and only allow responses from the server.

Consequently, a bidirectional connection is needed. Whereas a prototypical implementation used a legacy XHR Long Polling technique, the current version rests on the *WebSocket* protocol, compliant to RFC 6455. It allows an upgrade of an HTTP connection to a full-duplex channel with both sides capable of sending and receiving data at the same time. Nowadays almost all browsers support the WebSocket protocol and the sockets can be easily used from plain JavaScript without the need to install any additional libraries.

On connection establishment, the server offers a list of all available simulations to the client and provides it with progress updates (via broadcast). As soon as a selection is made, additional information on simulation structure and visualization initialization are given. The client constantly reports changes in its settings, movements and tick queries, the server replies with payload packets. These contain static positions, key- and delta frames and terrain tiles according to the OSM format. To save bandwidth, only the minimum of data required for rendering are transferred and further information are provided on separate request.

Because the messages are transferred as bytestreams, some sort of minimal header is needed for packet differentiation. For the sake of simplicity and efficiency, this header consists of a single byte which is prepended to every message and specifies its content. A byte allows to express the values from 0-255 and the following numbering scheme is used for disambiguation:

1xx	Client messages
2xx	Server messages
x**1**x	Control instructions
x**2**x	Visualization core data
x**3**x	Additional data
x**4**x	System and status messages

The subsequent depiction in figure 3.2 gives an overview on the packet types, their codes, responses and payloads. Please note that this diagram might be outdated due to changes or extensions during development. An up-to-date schematic and detailed descriptions (in German) can be found online in the documentation branch of the WebGLvis project repository.

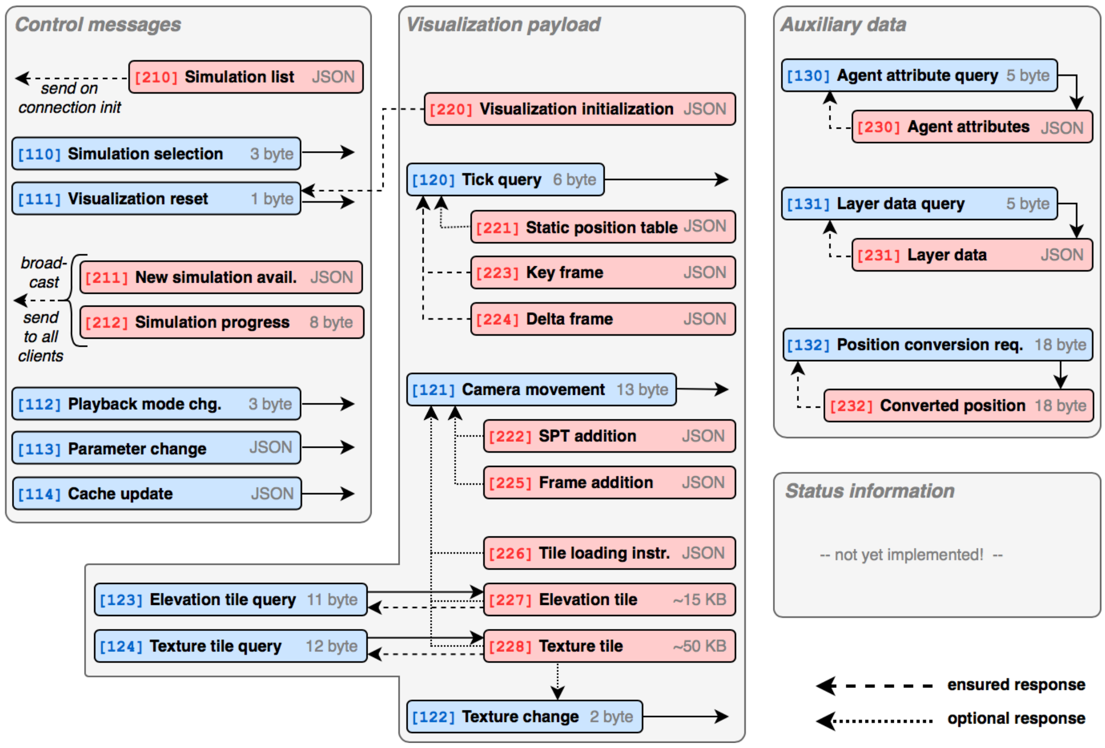

Control messages
[210] Simulation list JSON
send on connection init
[110] Simulation selection 3 byte
[111] Visualization reset 1 byte
broad-cast
[211] New simulation avail. JSON
send to all clients
[212] Simulation progress 8 byte
[112] Playback mode chg. 3 byte
[113] Parameter change JSON
[114] Cache update JSON
Visualization payload
[220] Visualization initialization JSON
[120] Tick query 6 byte
[221] Static position table JSON
[223] Key frame JSON
[224] Delta frame JSON
[121] Camera movement 13 byte
[222] SPT addition JSON
[225] Frame addition JSON
[226] Tile loading instr. JSON
[123] Elevation tile query 11 byte
[227] Elevation tile ~15 KB
[124] Texture tile query 12 byte
[228] Texture tile ~50 KB
[122] Texture change 2 byte
Auxiliary data
[130] Agent attribute query 5 byte
[230] Agent attributes JSON
[131] Layer data query 5 byte
[231] Layer data JSON
[132] Position conversion req. 18 byte
[232] Converted position 18 byte
Status information
-- not yet implemented! --
ensured response
optional response

Figure 3.2: Messages and responses of the visualization network protocol

4 Implementation

This chapter describes the implementation of the 3D visualization pipeline. It is based on the components elaborated in section 3.2 and refines them in order to integrate them into the MARS infrastructure and the MSaaS workflow.

4.1 Upstream Services

The visualization and output configuration takes place by means of two upstream services, which are deployed as part of MARS' automated cloud deployment: The *ReflectionService* does some background preprocessing on the simulation model, providing the necessary information for the *ResultConfigService* to show a configuration dialogue to the user.

Following the MARS architectural guidelines, both components are designed as *microservices*. Predominantly shaped by Fowler and Lewis (2016), this concept splits an application into a couple of separate services, where each service has its designated purpose. This allows for an independent development and adds interchangeability using a loose coupling via a common interface, often realized as HTTP REST calls (Masse (2011)). The diagram below shows an overview on the configuration stage, a detailed explanation is given on the next pages.

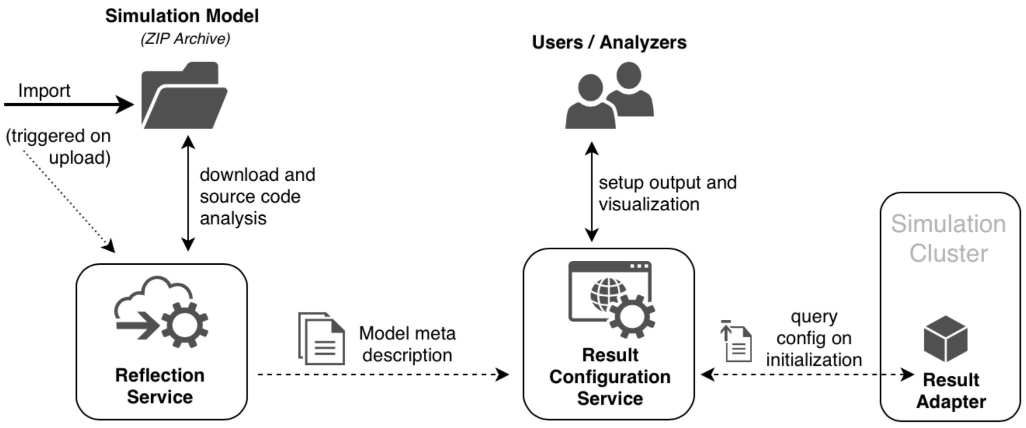


Figure 4.1: Overview: Import, analysis and result output configuration

4.1.1 *ReflectionService*

The necessity for the *ReflectionService* is owed to the fact that MARS is a generic multi-purpose simulation platform: Because the system's components are developed without any reference to a specific domain, MARS cannot "know" what type of model the user builds and of what entities and properties it contains. Still, this model-specific knowledge is anyway required to facilitate the configuration phases (e.g. the output configuration described in the next section has to know about the agent types and their attributes). Therefore a mechanism is needed to gather model intelligence upon request.

Responsible to obtain these essential data is the *ReflectionService*. As all microservices in the MARS ecosystem, it is a containerized application, built as a Docker image and run inside of a Kubernetes pod. Internally, the reflection service uses Microsoft's .NET Core 2.1 runtime, which is a modular and platform-independent framework for the C# programming language. The latter is also mandatory because all MARS models are compiled into that language.

The reflection service is linked to the model import and automatically triggered by the *FileService* on upload of a new model. For that purpose, it provides a REST-style API hosted by an inbuilt ASP.NET / Kestrel web server. This endpoint is cluster-internally available and expects a model identifier as parameter. On invocation, the referenced model archive is first downloaded and then extracted into container memory. These actions are conducted by the main execution thread that also runs the API.

After extraction, the reflection service performs a code analysis on the model's C# binaries in order to detect all simulation-related types. This reverse-engineering technique of looking into compiled code and extracting the type names, headers and signatures is called 'reflection', hence the service's name. For reasons of dynamic DLL loading and unloading, the reflection has to be done in an isolated process for each model. These child processes are spawned and orchestrated by the main (API) thread, the inputs and results are piped using data stream redirection.

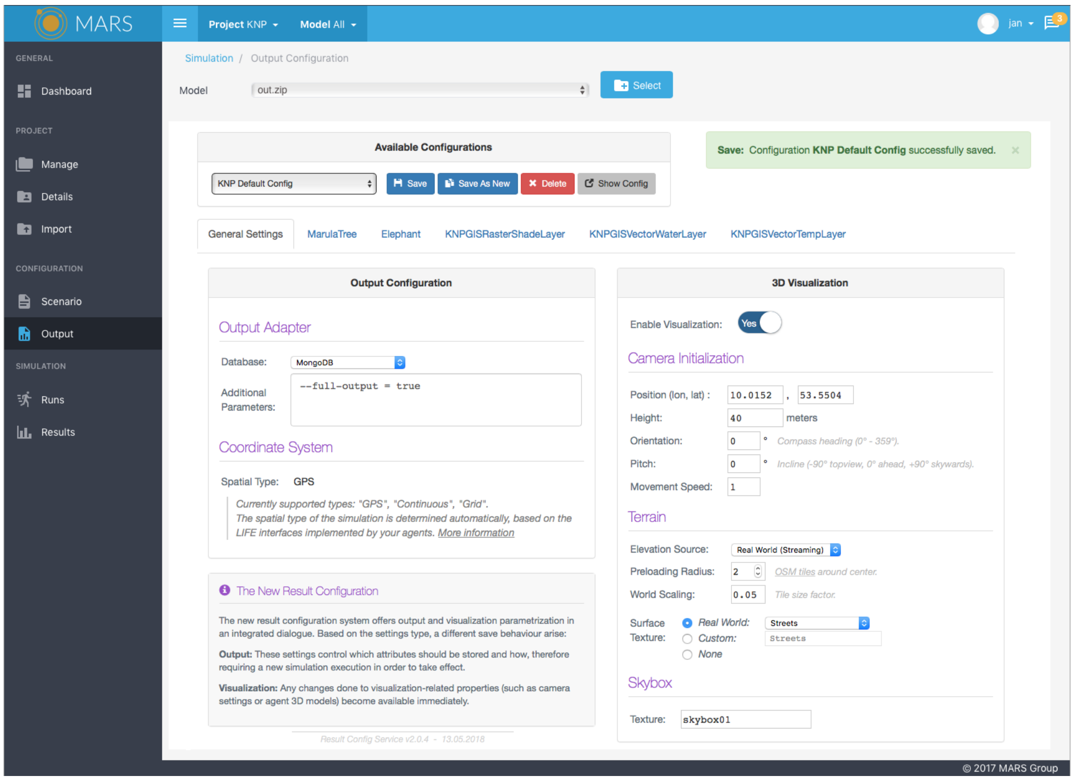
Inside the reflection process, all DLLs are crawled for MARS-related interfaces in order to find the model's agent and layer type definitions. For every type, its properties (public attributes expressing the externally observable state), the constructor signature (parameters required for initialization) and the inheritance hierarchy (appropriate creation order) are analyzed. After the reflection has passed for all libraries, the parser result is forwarded as a JSON-formatted meta description to the *MetadataService*, which stores it alongside with the model's other information in the database. Building on this, the subsequently detailed *ResultConfigService* has the necessary information to provide a custom-tailored configuration dialogue to the user.

4.1.2 *ResultConfigService*

Before a simulation can be run, the user has to decide what output data they want to collect during execution. As mentioned in section 3.3.1, the full output of *everything* is in the vast majority of cases not feasible and for that reason, the output configuration was incorporated into the MARS MSaaS workflow. Also, the 3D visualization needs several parameters (e.g. what to visualize and which 3D models to use) and an interface where this configuration takes place.

Both jobs are accomplished by *ResultConfigService*; another microservice running containerized in the MARS cloud. It provides a server-side endpoint for configuration management and a web interface to display the appropriate dialogues to the user. The screenshot 4.2 at the bottom of this page gives an impression of this UI.

A result configuration comprises of two distinct files, an output and a visualization configuration, which may be separately updated and changed. The visualization config can be changed anytime and immediately takes effect, this means it is possible to change the visualization defaults for a saved simulation run. Output config changes on the other hand naturally require a new simulation execution in order to be incorporated.



Model: out.zip

Available Configurations: KNP Default Config

Save: Configuration **KNP Default Config** successfully saved.

Output Configuration

Output Adapter

Database: MongoDB

Additional Parameters: `--full-output = true`

Coordinate System

Spatial Type: GPS

Currently supported types: "GPS", "Continuous", "Grid". The spatial type of the simulation is determined automatically, based on the LIFE interfaces implemented by your agents. More information

The New Result Configuration

The new result configuration system offers output and visualization parametrization in an integrated dialogue. Based on the settings type, a different save behaviour arise:

Output: These settings control which attributes should be stored and how, therefore requiring a new simulation execution in order to take effect.

Visualization: Any changes done to visualization-related properties (such as camera settings or agent 3D models) become available immediately.

Result Config Service v2.0.4 - 13.05.2018

3D Visualization

Enable Visualization: Yes

Camera Initialization

Position (lon, lat) : 10.0152 , 53.5504

Height: 40 meters

Orientation: 0 ° *Compass heading (0° - 359°).*

Pitch: 0 ° *Incline (-90° topview, 0° ahead, +90° skywards).*

Movement Speed: 1

Terrain

Elevation Source: Real World (Streaming)

Preloading Radius: 2 *OSM tiles around center.*

World Scaling: 0.05 *Tile size factor.*

Surface Texture: [✓ *Real World:*] Streets / *Custom:* Streets / *None*

Skybox

Texture: skybox01

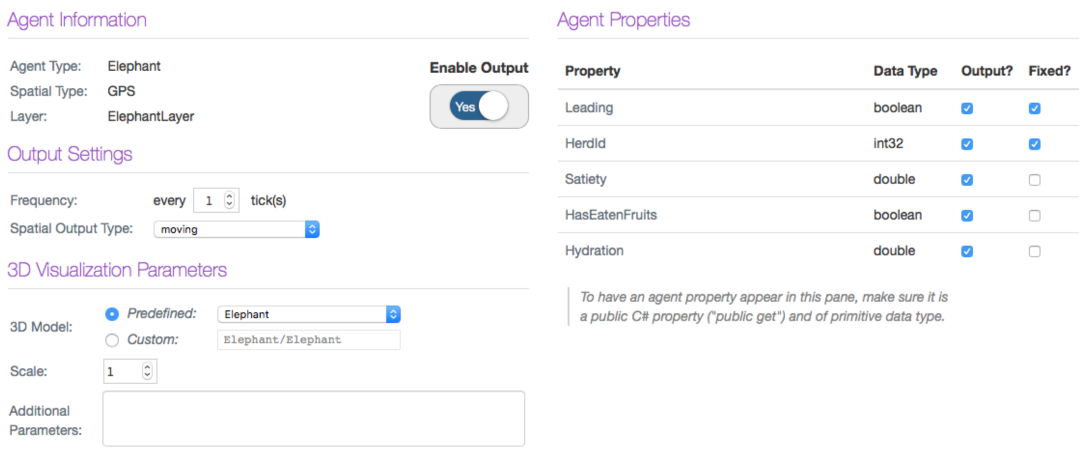
© 2017 MARS Group

Figure 4.2: User interface for the output & visualization configuration

The backend for the result configuration service is also running on the .NET Core 2.1 platform and it provides a REST API supporting CRUD operations on result configs. It is linked to the *MetadataService* to query a model's meta information with the structural description (that were the data obtained by the ReflectionService) and generates a suitable config with preset values upon first call. Existing configurations are stored in a *MongoDB* database, for which the common MARS config instance is used. The database is also utilized as an update system, using capped collections and tailable cursors. This way, the configuration consumers (output adapter and visualization service) can subscribe and react for changes – e.g. the visualization can perform a reload in order to auto-apply camera initialization values.

The frontend is implemented as an Angular 4 app and integrated into the MARS web suite, directly accessible from the side panel which depicts the MSaaS workflow. Because of their logical connexion, both configurations are displayed in a common UI and are partitioned into a general settings section and type-specific preferences. The *General Settings* tab contains an output column that allows to set the database target and optional parameterization, as well as several visualization defaults. These are type-overarching settings, like camera position, world and movement scaling, the terrain setup (texture, elevation source and preloading settings) and several augmentation features, e.g. a background skybox.

Next to these common settings are tabs for the type-specific configuration (one tab per type). Screenshot 4.3 displays such a type-specific configuration dialogue, in this case for an elephant agent. The blue switch button governs whether the agent type should be output or not, the settings below control the output frequency and the spatial type (moving or stationary). Visualization-related settings follow afterwards and allow 3D model and scale specifications. The right pane lists all available properties and here the user can make their output selection.



Agent Information

Agent Type:	Elephant
Spatial Type:	GPS
Layer:	ElephantLayer

Enable Output: Yes

Output Settings

Frequency:	every 1 tick(s)
Spatial Output Type:	moving

3D Visualization Parameters

3D Model:	[✓ Predefined: Elephant] Custom: Elephant/Elephant
Scale:	1
Additional Parameters:	

Agent Properties

Property	Data Type	Output?	Fixed?
Leading	boolean	✓	✓
HerdId	int32	✓	✓
Satiety	double	✓	
HasEatenFruits	boolean	✓	
Hydration	double	✓	

To have an agent property appear in this pane, make sure it is a public C# property ("public get") and of primitive data type.

Figure 4.3: Agent type-specific output and display settings

4.2 Simulation Output

After the simulation setup services were discussed, this section now proceeds with describing the actual output and storage components. It introduces the *ResultAdapter*, which is the part of the simulation engine responsible for output elicitation and write out. Following this, the database scheme and the storage & retrieval algorithms are presented.

4.2.1 *LIFE ResultAdapter*

Rooted within the simulation engine *LIFE*, the *ResultAdapter* is the component that manages the data acquisition and its storage in the result database. Like the other engine parts, it is realized as a .NET Core project, written in the C# language and compiled together with the rest of the system into a single execution unit. The output logic is triggered from the main simulation loop, which is located in the *RTEManager*. This loop consists of an encapsulated three-tier routine (for every simulation step / for every layer / for every agent) that calls the result adapter after each completed tick and for each simulation entity. Additional calls are performed for the agents spawned or deleted during tick execution.

Before the simulation output can begin, the result adapter needs to be initialized with a suitable result configuration (as explained in section 4.1.2). When LIFE is started, it gets several parameters supplied, amongst other things the result configuration ID. This identifier is passed along to the result adapter, which downloads the respective file from the *ResultConfigService* and parses its JSON structure. With the containing information, the result adapter knows about all agent and layer types, their attributes and the user's desired output. It still has no output capabilities yet, because the output registration function for the simulation entities has to be defined at compile-time, so it can only assume the vague *ITickClient* interface (which just says that an entity it executable).

To overcome this gap, the knowledge about the concrete types obtained from the result config has to be injected in an executable fashion. This is done using on-the-fly code generation as supposed in section 3.3.2. A *logger generator* reads the simulation structure and creates output modules for all agent and layer types by writing source code. Every simulation type gets its own class, taking the names and datatypes of the actual properties into account, as well as the user's output wishes. All *loggers* are unified under a common output interface, demanding the following functionality (see savings strategy presented in 3.3.1):

- *GetMetatableEntry():* Meta structure with agent information and immutable properties.
- *GetKeyFrame():* Complete output with all selected agent states for the current tick.
- *GetDeltaFrame():* Output containing only states changed since the last tick.

These methods are filled with the type-specific implementations, thus allowing to output the respective entity type. All class definitions are pooled in a single C# source code file, which is compiled using the .NET compiler platform Roslyn into executable binary code. The compiled DLL is then loaded and injected into the runtime, making the new logger types available for instantiation. This way, the generic core of the *ResultAdapter* is able to enhance itself with the output capabilities needed to write out user-specific simulation types. Figure 4.4 illustrates this initialization process and the further usage:

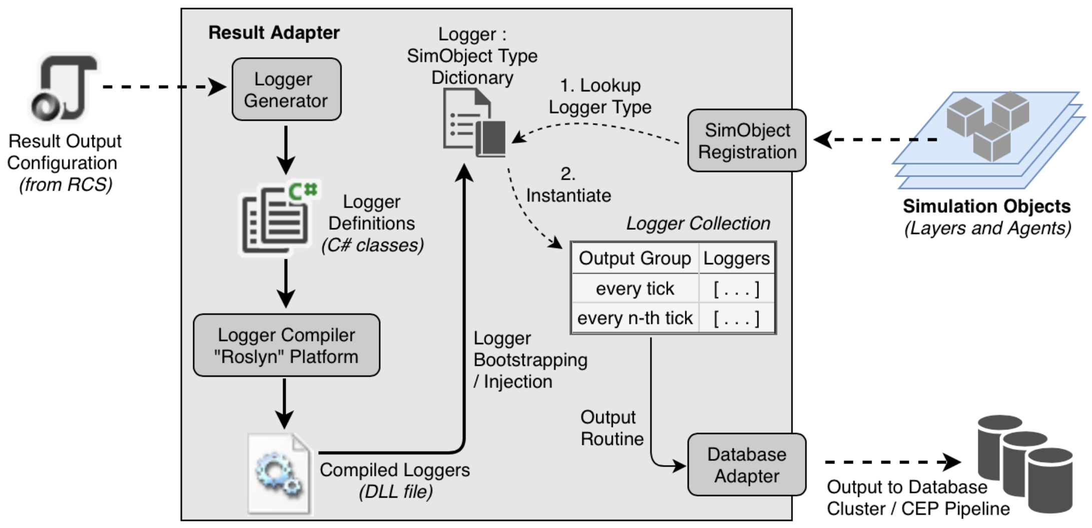

Result Adapter
Logger Generator
Result Output Configuration (from RCS)
Logger : SimObject Type Dictionary
1. Lookup Logger Type
SimObject Registration
Simulation Objects (Layers and Agents)
Logger Definitions (C# classes)
2. Instantiate
Logger Collection
Output Group | Loggers
every tick | [. . .]
every n-th tick | [. . .]
Logger Compiler "Roslyn" Platform
Logger Bootstrapping / Injection
Output Routine
Database Adapter
Compiled Loggers (DLL file)
Output to Database Cluster / CEP Pipeline

Figure 4.4: The simulation output system

The injected logger templates are stored in a mapping (C#: *dictionary*), establishing a relation *simulation type* $\rightarrow$ *logger type* for all simulation entities to expect. This mapping is consulted every time a new entity is created and registered at the result adapter (mainly during simulation bootstrapping, in case of agents also possible during runtime) to lookup and instantiate a corresponding logger. The created loggers are managed in a collection, categorized into groups according to output frequency.

The actual output logic happens in a routine that is triggered after each simulation step. First, the meta information of all newly added entities are written to the database, afterwards it loops in parallel over all active output groups and the loggers they contain. For every logger, either a key- or delta frame (see 4.2.2) is produced. These output packets are accumulated and batch-written into a data sink, which can be either a single database or a complex event stream processing pipeline. In addition, a notification is sent to inform potential output consumers (e.g. the visualization backend) about the availability of new data.

4.2.2 Database & Storage Formats

MARS uses the NoSQL database MongoDB (from "humongous", referring to its scalability and flexibility) for the storage of both configurations and simulation results. Unlike relational database such as MySQL, the NoSQL approach goes without fixed tabular relations and uses other data structures, e.g. key-value or document-based storage. This offers a greater flexibility and, thus, is well suited for the storage of arbitrary simulation results. With regard to the MongoDB, a database is partitioned into a set of *collections* which hold the data in form of JSON objects. These objects use the common key-value syntax to hierarchize and express the information.

In place of a MongoDB, it is also possible to use MARS with a complex event processing (CEP) pipeline. The ecosystem provides two services, Kafka (a data streaming platform, used as immediate data sink) and Cassandra (a resilient and distributed database for *huge* data, used as persistency layer) for that cause. Especially large-scale simulations with a very high data output can benefit from such a two-part storage solution.

From the *ResultAdapter*'s perspective, the write-out target is transparent to the output routine and it provides implementations for both use cases. What output target shall be used can be set in the result configuration dialogue and this setting is taken into account during result adapter initialization. For most simulations, the default configuration employing a MongoDB is used. For large simulations, this database is distinct from the config database and gets distributed (in MongoDB terminology: *sharded*) across multiple instances in the cluster. The experiments described in this thesis, however, use a single instance in favor of simplicity, because data throughput performance is neglectable.

As mentioned above, NoSQL databases use *collections* instead of tables as the unit of organization. The result database has a **SimulationRuns** collection, that keeps track of all performed simulation runs. For every run an entry is made, telling about the model and configuration specifics, the layer and agent hierarchies, key frame ratio, tick counter and simulation progress. This collection gets updated as the simulation progresses and is the primary information source for all MARS evaluation and visualization tools.

For the agent output, the result adapter differentiates between three output formats: *Meta entries*, *key frames* and *delta frames* (see the interface in 4.2.1). These formats pick up the storage savings idea from 3.3.1 and are stored in the following collections on a per-run basis:

Meta description table (<*SimID*>-meta)

Though actually a collection, this structure can be imagined as a table, holding entries for every agent that was present in the simulation run. It contains the agent identifier and its type, the layer it resides on, creation and deletion tick info and all of its static properties. If the agent is stationary, also its position and orientation are stored here.

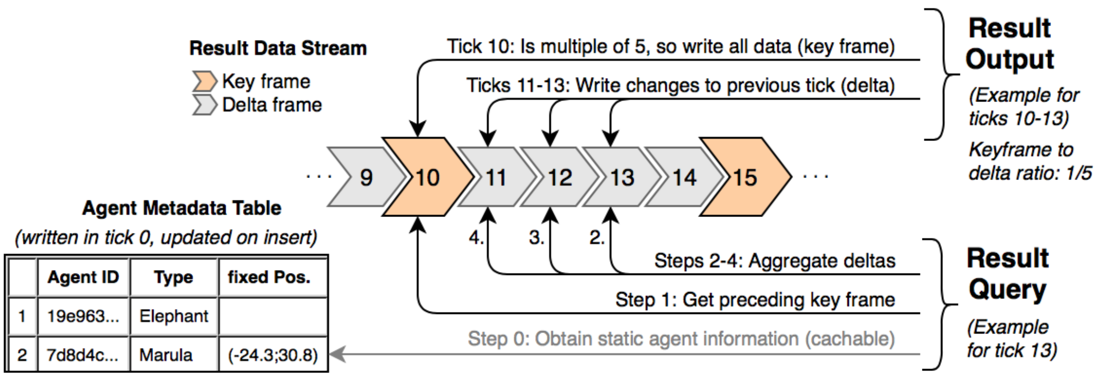
Key frame collection (<*SimID*>-kf)
This collection holds a series of key frames for every agent. A key frame is a full-state output of all varying agent attributes, i.e. the position data for moving agents and all properties flagged as non-static. The key frames are organized and retrieved by using the agent identifier and frame index number.

Delta frame collection (<*SimID*>-delta) (optional)
The delta frame table is in terms of structure similar to the key frame table, but instead of a complete specification it only contains the attributes changed since the last tick. This collection is optional and can be omitted in case no deltas are used.

In order to write and read the output of an agent for a given tick, the distribution over the above-mentioned collections has to be kept in mind. The meta entry is written once, updated on agent deletion and can be cached from the reader's side. For the varying properties, the key frame interval K has to be taken into account to decide whether a full-state and delta output is required. When reading, the frame F for a tick T is obtained by using the following formula:

$$F_T = KF_{T/K} + \sum_{i=K\cdot(T/K)+1}^{T} DF_i$$

with KF and DF being an entry in the key- respectively delta frame table. Figure 4.5 shows an example query for tick T=13 with an interval of K=5. The complete frame F_{13} is retrieved by getting $KF(2)$ and accumulating the deltas for tick 13, 12 and 11.


Result Data Stream
Key frame
Delta frame
Tick 10: Is multiple of 5, so write all data (key frame)
Ticks 11-13: Write changes to previous tick (delta)
Result Output
(Example for ticks 10-13)
Keyframe to delta ratio: 1/5
9
10
11
12
13
14
15
4.
3.
2.
Steps 2-4: Aggregate deltas
Step 1: Get preceding key frame
Step 0: Obtain static agent information (cachable)
Result Query
(Example for tick 13)
Agent Metadata Table
(written in tick 0, updated on insert)

	Agent ID	Type	fixed Pos.
1	19e963...	Elephant	
2	7d8d4c...	Marula	(-24.3;30.8)

Figure 4.5: Write and read operations for the tick-based storage formats

4.3 The Visualization Service

Resting on the configuration done in 4.1.2 and the results saved in 4.2.1, this section presents the visualization back-end service responsible for data aggregation and provisioning. In its basics, it consists of a web endpoint for client connections, accessors for the data sources and notifications and a middleware in between, responsible to handle and answer the clients' queries. This overview is depicted in figure 4.6, detailed explanations follow afterwards.

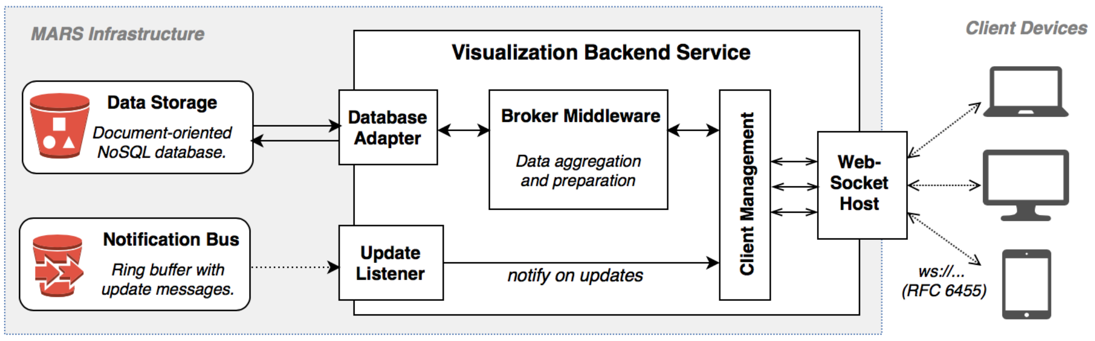
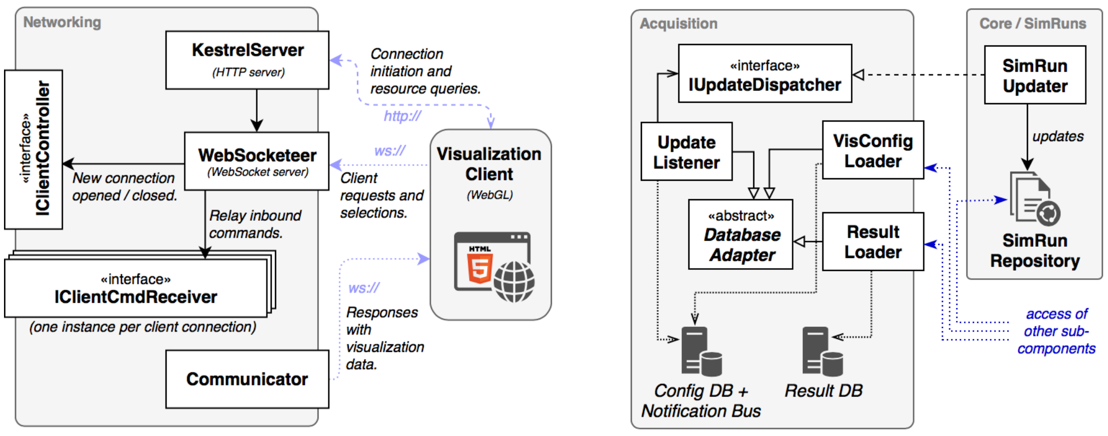


Figure 4.6: Components of the visualization back-end

4.3.1 Client Connection Handling

The visualization service is an application running on .NET Core 2.1, both stand-alone executable and containerizable for cloud deployment. It employs the ASP.NET in-built "Kestrel" server to provide HTTP- and WebSocket endpoints. HTTP is used for the initial negotiation and for the asset downloads, including the resources for the user interface (HTML, CSS, JS) and – on visualization start – the 3D models to be displayed. After connection establishment, the client performs an upgrade to a full-duplex WebSocket connection to allow bidirectional traffic, as detailed in section 3.3.4. WebSocket connection handling is done by the *WebSocketeer*, a custom module injected into the Kestrel server and called on WebSocket packet reception. It controls the active handles, deserializes the inbound traffic (1xx codes) and delegates the queries and instructions to the respective client control module.

Every time a new client connects to the back-end, an own independent representation is created to remember the simulation selection, camera position, viewing range and playback mode of that client. This proxy has access to the service-internal submodules for scenario-, agent-, terrain- and auxiliary data retrieval and parameterizes them with the properties of its client in order to deliver custom-tailored query responses. The forwarding of the queries from the WebSocket handler to the proxy requires the latter to implement the *IClientCmdReceiver* interface which is registered as query processor in the WebSocket handler and mapped to that client. To send messages, every client representation has its own *Communicator*, created by the WebSocket handler and set up to communicate over the negotiated channel. It offers interfaces to send the 2xx packets and is used to transmit the replies to the front-end visualization. Figure 4.7 shows the networking internals (left side):



Networking
KestrelServer
(HTTP server)
«interface» IClientController
WebSocketeer
(WebSocket server)
New connection opened / closed.
Relay inbound commands.
«interface» IClientCmdReceiver
(one instance per client connection)
Communicator
Connection initiation and resource queries.
http://
ws://
Client requests and selections.
Visualization Client
(WebGL)
ws://
Responses with visualization data.
Acquisition
«interface» IUpdateDispatcher
Update Listener
VisConfig Loader
«abstract» Database Adapter
Result Loader
Config DB + Notification Bus
Result DB
Core / SimRuns
SimRun Updater
updates
SimRun Repository
access of other sub-components

Figure 4.7: Connection handling (left) & data acquisition (right)

4.3.2 Data Acquisition and Stockpiling

For the data retrieval the visualization service has multiple connectors which tap into MARS' databases. These operations are encapsulated in the acquisition plane, providing a simple query API to the other subcomponents. At the current stage only MongoDB instances are used, resulting in a common database adapter and three concrete implementations. During initialization, the back-end establishes connections to the result database, the configuration store and the notification bus. It queries all available simulations and maintains them in an internal structure, ready to be delivered to connecting clients. Afterwards the acquisition plane works on an on-demand basis, as depicted in figure 4.7.

The first and foremost task is of course to load the simulation results by accessing the key and delta frame collections in the result database. Besides, also the configuration database is accessed to load the visualization configs. Whenever a client selects a new simulation, the database adapter checks first if the visualization init config is already in memory and otherwise downloads it. Henceforth, this config is provided with updates from the notification system.

Via the MongoDB also several events are propagated: Simulation starts, progresses and configuration updates are announced as short messages in a capped collection. The visualization service subscribes to this notification bus by means of a tailable cursor and updates its internal simulation list, agent metatables and cached configurations on an update announcement. Changes regarding the simulation list are automatically broadcasted to the attached clients via code 211/212 messages, new visualization defaults require a client-side reset.

4.3.3 Provisioning of Terrain Assets

The provisioning of terrain data is an essential part for the visualization because it helps to reconcile the simulation entities with the real-world and eases user orientation. To this end, the visualization service supports multiple terrain sources. If a custom GIS elevation file is used in the simulation, this file shall later be loadable for the visualization *(not yet implemented)*. In case no elevation is used but a GPS reference exists and the user desires to project the result space to earth terrain, the MapBox API is accessed to facilitate streaming of real-world heightmaps and textures. Or, if the terrain is irrelevant (e.g. for a simple, grid-based simulation), also a flat plane can be used, optionally with a texture overlayed.

As map tile reference system, the visualization service uses the OSM tiling for real-world terrain data or a custom, OSM-compatible system for other sources. In order to create an unifying abstraction from the different data sources so that the origin is transparent for the client-side visualization, a coordinate conversion service is offered. For every client, an own *TileCalculator* gets instantiated which is tied to the selected simulation and converts between the camera- and simulation coordinate space. Internally, it maps between the GPS or self-chosen cartesian system, the various zoom levels and the 3D engine coordinates by taking a scaling factor into account, which can be set during visualization configuration.

4.3.4 Map Tile Caching

Because the terrain streaming from external data sources may impose a bottleneck, a tile cache is integrated into the visualization service. It helps to reduce the network load and improves response time significantly by saving real-world elevation and texture tiles on the server's hard disk. Only on cache miss, the file has to be downloaded first via streaming API and otherwise can be immediately returned.

Regardless whether the service in run on a local machine or in the cluster, memory is a finite resource and thus a deletion mechanism for unneeded tiles is needed. This job is done by a cache manager that tracks all stored files and the accesses to them. On program startup a maximum cache size is stated (by default 350 MB) and this limit is enforced by the cache manager by means of two deletion policies: The first and default option is the LRU (least-recently used) policy which looks upon the time of last access for a file. It lists all cache files in a table, sorted in ascending order by the timestamp of last retrieval. If a new file has to be downloaded and the total cache size exceeds the memory threshold, the topmost entries are removed until the limit is fulfilled again. As a second option, the LU (less used) policy may be chosen which acts in accordance to the utilization of a terrain tile and deletes the file with the lowest access counter. This may lead to an undesired behavior, because new files would be deleted again almost immediately. As a remedial action, an immunity flag is set to prevent the deletion of a just downloaded file for a limited time span. By default, this interval is set to 10 minutes of immunity.

In order to keep the cache consistent between executions, it needs some kind of persistency to retain the entries and their properties. For that purpose a text file is used that has a tabular structure and contains all files present on disk and their caching-relevant attributes (access time, counter, path, file size and immunity flag). This table is sorted according to the current deletion policy to allow a fast resumption on the next program start. Its contents are updated in regular intervals and on service shutdown.

4.3.5 Logging & Configuration

To allow traceability and debugging, a logging system is incorporated into all components that can be used to write messages to the standard output (console) and to a log file. Five logging levels are available to specify the criticality of the output; they range from critical *errors*, *exceptions* and *warnings* to *informative messages* and *tracing output*. For both logging targets the importance thresholds are set on a sub-component basis (system, networking, client management...) during startup in the *Logger* service, which then is used throughout the program. This logger is accessible from anywhere and log messages can be posted by calling the logger with the message content and its log level.

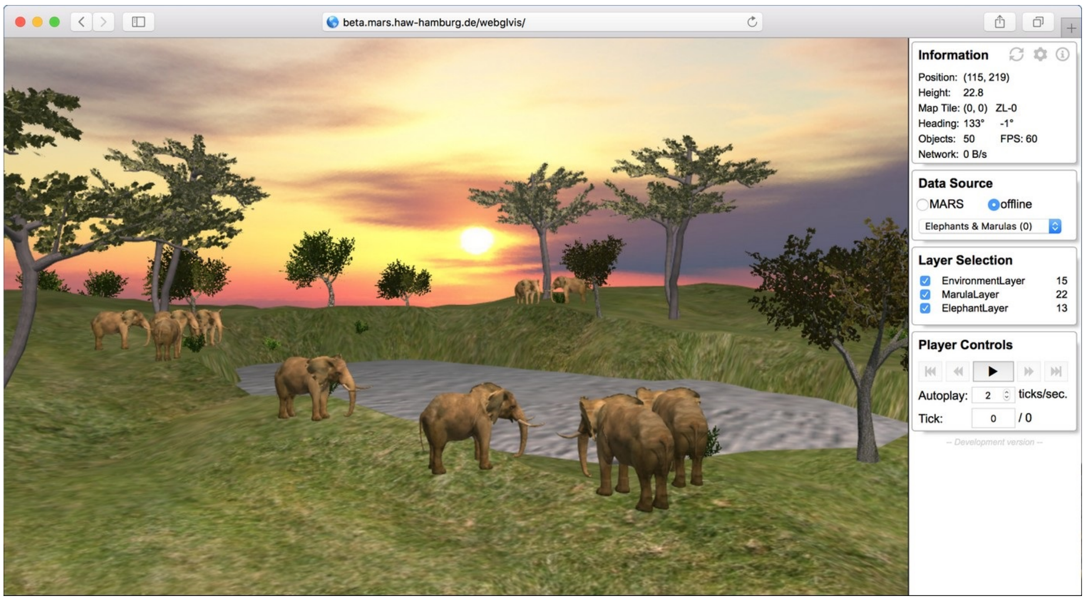
With its many parameters, the visualization service offers a broad basis for configurations: Data base paths, logging levels, hosting ports and terrain cache thresholds can be adjusted to fit the current environment. All these options are preset with default values and may be overwritten by means of a configuration file: The *ConfigReader* looks in the working directory for a *visconf.ini* file, which contains the parameters to be overwritten. If such a file is found, it is parsed and the defaults are substituted with the customized settings. The config file uses a simple $key = value$ syntax, a complete reference of all parameters and their valid assignments may be found in the repository. Please note that changes to the configuration file require a program restart in order to take effect.

4.4 *WebGLvis* – the Front-End 3D Visualization Client

Last but not least to the actual visualization: After the previous sections discussed the configuration, output and aggregation stages, this one presents *WebGLvis*, the final link in the chain. It is a 3D visualization based on WebGL technology and renders the simulations results in the web browser. This section first describes the user interface and how it is used, before it then proceeds with a detailed description of design decisions and realization.

4.4.1 Interface Layout and Usage

The subsequent screenshot 4.8 provides an impression of the *WebGLvis* visualization. It displays an elephant herd gathered around a water hole from a savannah model.



Information

Position: (115, 219)
Height: 22.8
Map Tile: (0, 0) ZL-0
Heading: 133° -1°
Objects: 50 FPS: 60
Network: 0 B/s

Data Source

MARS [✓ offline]

Elephants & Marulas (0)

Layer Selection

✓	EnvironmentLayer	15
✓	MarulaLayer	22
✓	ElephantLayer	13

Player Controls

Autoplay: 2 ticks/sec.
Tick: 0 / 0

-- Development version --

Figure 4.8: Screenshot of the *WebGLvis* 3D visualization

As seen above, all user interface controls are clearly arranged one below the other at the right side of the screen. The box on top displays the current camera position and orientation as well as a menu for visualization-specific settings, like lighting, view range or manual positioning. Below follows the simulation selection drop-down. When the user makes a choice the visualization initializes itself to the presets, which were defined in the configuration stage (see 4.1.2) and can be updated any time. The camera gets positioned accordingly and the specified start tick is loaded (not necessarily but by default the tick 0). Also the next two UI elements become active: The first selection lists all layers of the simulation together with the number of agents contained on them and a checkbox in order to activate or deactivate the visualization of that layer. Beneath the layer section follow the playback controls. They have a structure similar to the control panel of a video player and consist of four buttons which allow the automatic stepping through the simulation (play/pause), manual navigation (one tick forward/back) and jumps to the first respectively last tick. It is also possible to directly specify the tick to render and to set the playback speed.

The camera movement in the 3D scene is made using the arrow keys or – as it is common in many games – using W,A,S,D with the option for sideways movement by means of Q and E. The height can be adjusted with the mouse wheel or alternatively with the +/- keys. For camera panning the mouse is used with the right mouse button pressed down. Apart from that, the mouse cursor can be moved freely in order to interact with the interface or for agent selection. An agent may be focussed with a left click; it then gets highlighted in the visualization and its additional properties are queried from the back-end. As soon as the response is received a small tooltip window shows up in the lower right, displaying the agent's attributes. This window gets automatically updated on tick changes and queries the new information independently.

4.4.2 Design Decisions

At the beginning, the initial plan was to build the 3D visualization client with the popular game engine Unity3D. After a few experiments, however, this attempt was abandoned: Even though it may be tempting to utilize an extensive and well-known game engine, a number of problems arise in this particular context. Unity comes with a full-fledged IDE, which offers a lot of powerful features for game design, but also craves for a profound training in order to be productive. It puts several demands on the internal structuring (e.g. the way how game objects are built and the UI is designed) and saves the project in proprietary formats only editable by Unity. A lot of the provided functions are not needed, because in this use-case we have no game logic and just want is to bulk-load our entities which were preprocessed by the back-end into the game loop to get them rendered.

Another problem is posed by the build pipeline: As it is common for games, Unity's main compile targets are platform-specific stand-alone clients. This contradicts the requirements from section 3.1. While there is a WebGL export functionality, at the time of evaluation it only worked for the *Firefox*. Furthermore, the build pipeline takes quite some time (5+ minutes) and performs numerous compile steps from C# via assembler to minified JavaScript. Because several effects only occur in the production build and not in the IDE's preview runtime, the code is very tedious to debug, which makes the overall development a time consuming and frustrating endeavor.

Although the 3D engine is a vital part, it is also just one component among many others. Apart from the rendering, an input handling is needed, networking, an internal world representation, model loaders and a lot more. For these reasons it was deemed appropriate not to focus too much on the engine itself and that a self-developed WebGL engine – limited to the essential features, performant and proven in previous applications – should be more than sufficient for a first draft. A reasonable encapsulation assumed, the engine may be easily replaced with a more promising solution in the future (see outlook 7.3).

Another goal is the ease-of-use, realized by means of a concise and simple user interface. This interface shall be integratable into the MARS WebUI and should still be independently accessible to keep the development process fast and easy. For these reasons it was decided to refrain from utilizing sophisticated web application frameworks, such as *Angular* or *React*. Indeed, the visualization has no external dependencies, apart from the (embedded) high performance math library glMatrix used for OpenGL vector and matrix calculations and the Icomoon font builder for the interface icons.

As a result, the visualization is a complete self-development, built on pure HTML5, CSS3 and JavaScript (ES5). It runs in all common browsers and thanks to the absence of third-party modules, its core is very lightweight. The custom engine has several performance optimizations and delivers enough power to run visualizations also on older machines. On deployment, a custom compressor joins all visualization code in a single JavaScript redistributable with less than 80KB of size. This allows the visualization core to load and initialize almost instantly, even on slower connections.

4.4.3 Realization

On file level, the visualization consists of an HTML document, a style sheet, an icon font and a set of JavaScript classes that form the actual program. All classes are instantiated and linked together in an initialization function, which is triggered automatically as soon as the user opens up the visualization. The application itself is structured according to the *three-tier-architecture*, a widespread software design pattern that divides a program's components into presentation, application logic and data layer.

In this case the lowest (level 1) tier is the provisioning layer. It contains the WebSocket adapter that automatically connects to the visualization back-end during initialization. This connector encapsulates the transmit and receive functionality by incorporating the network protocol presented in section 3.3.4. To facilitate a loose coupling, the adapter works on the publish-subscribe principle and maintains lists for all message types received. By subscribing to these lists, all further components may register for those packets relevant for them and get called back via delegate invocation on message reception.

The opposite tier is the user interface (level 3). The structure and arrangement of the interface controls is expressed in the HTML, the CSS describes their appearance and on JavaScript side the behavior is specified. To allow here a good modularity and separation of concerns as well, every UI element has its own controller class. The controllers take care of their display elements and evaluate the user input. For the purpose of communication with the middle tier, two facades are used: The *QueryRelay* serves as gateway for the downstream traffic and has access to the relevant core components. It is exposed to the controllers and performs the respective calls on the logic layer to put the user interactions into practice. For the reverse direction the *ResultForwarder* is used. It is registered in the output-emitting core components and in turn has access to the UI controller in order to pass changes back upwards.

In the middle between data access plane and UI is the core layer (tier 2). It performs all the asset management as well as the actual rendering. In its heart, the *WebGLEngineV3* is located. It is a simple, yet performant 3D engine that uses the WebGL API. WebGL is an open and cross-platform standard to make OpenGL ES 2.0 calls accessible for JavaScript and as such allows hardware-accelerated rendering from within the browser. Drawing is done using fragment (2D) and vertex (3D) GLSL shaders which are compiled and uploaded to the GPU. The engine comes with inbuilt shaders, but also can be parametrized from the outside.

The engine's main loop is a four-stage rendering routine: First the skybox is drawn, followed by the terrain. To increase performance, the terrain tiles' meshes are drawn as triangle strips. Afterwards the agents and optional "doodads" (decoration objects to embellish the world) are rendered. These entities are grouped by 3D model, so that the model is loaded once and then reused for all objects. For that purpose the engine maintains render groups for the world assets. On user left-click, the rendering loop executes a fourth stage, which draws a color map to an offscreen buffer with RGBA-encoded entity identifiers. This allows agent de-referencing by reading the color at the cursor position and comparing it with the offscreen VBI's.

In order to rasterize the 3D space onto the 2D screen, the engine needs the *CameraController*. This class is responsible for the camera positioning and view cone projection. It uses the *glMatrix* library to calculate the model-view and projection matrices based on the current position and orientation. Also related to the engine are the 3D structures and the *ModelLoader*: The latter keeps track of all 3D assets in use and also fetches new models on demand. On first use, the loader retrieves the model file asynchronously via HTTP and loads all vertices, vectors and textures into OpenGL array buffers. Internally, multiple types of scene components are used for representation, their hierarchy is shown in figure 4.9.

The 3D models are stored in a *.m4l* file, a custom format that features a JSON-based structure to allow an easy parsing. It is a merge of *OBJ/MTL* and inspired by *Blizzard's MDX/M3* formats, supporting submeshes, bones and animations. An broad selection of 3D models is available and can be chosen during visualization config, the user-upload and conversion of additional models is in principle possible (see outlook 7.3).

During runtime, all loaded entities are contained in the *World* class. They are managed by a set of *display modules*, with every module having its designated task. Currently visualizers for agents and the environment exist (layers as stub). These modules register for the respective messages and create, modify and delete the 3D objects accordingly. The terrain is constructed in the environment visualizer and uses a threefold loading mechanism. After a tile load instruction was received, the visualizer inserts a (neighbor-aligned) flat tile and adds elevation and textures when they become available. The zoom levels are managed in a hierarchy tree with transitions to active rendering taking place as soon as all subtiles have been completely loaded.

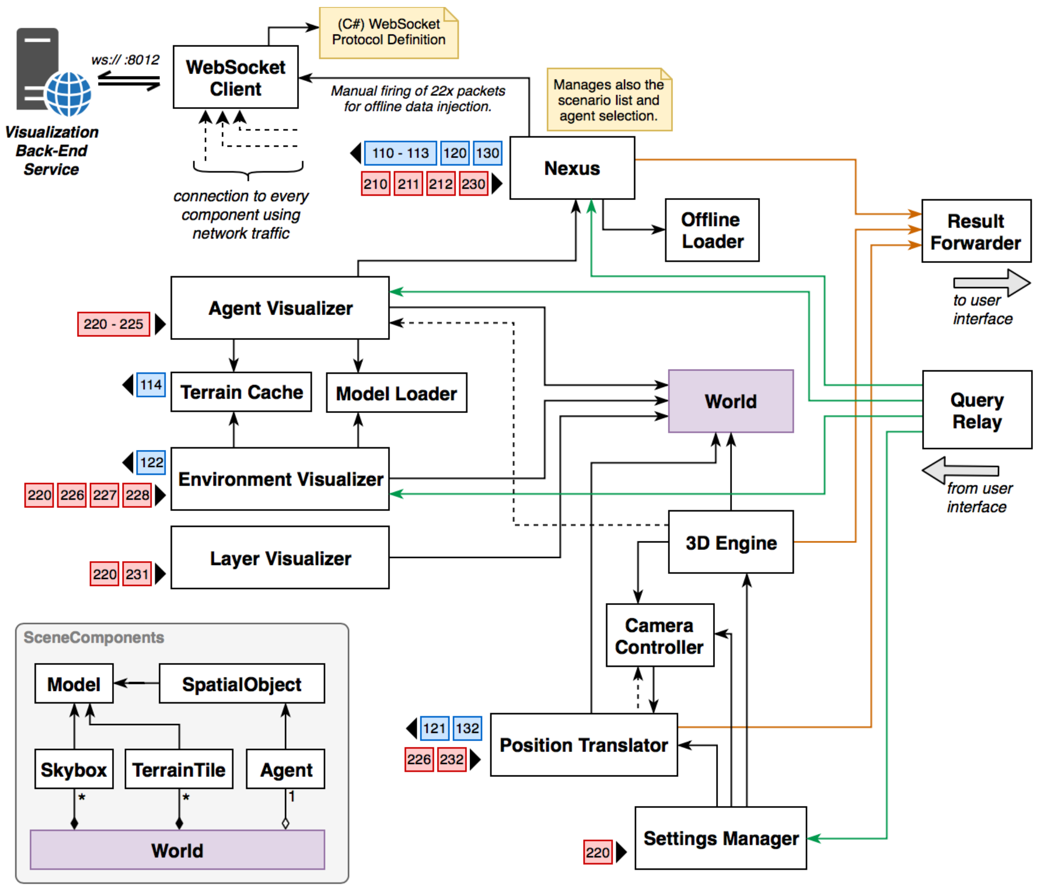


Figure 4.9: Internal structure of the visualization client

5 Experiments

In order to investigate the hypotheses claimed in section 1.4.1, this chapter presents several experiments. It begins with some notes on the common setup by introducing the reference model, its parametrization and runtime environment and then details the individual experiments.

5.1 Setup

5.1.1 The Reference Model

Basis for the scalability experiments is the *Wolves and Sheep* model, a simple predator-prey scenario which runs "out of the box" at an arbitrary scale and without any external data sources required. It is a derivative of the showcase model built for the *MARS-Fest* held at the University of Florida in May 2017 and consists of simple reflex agents which use the SRA cycle mentioned in Dalski (2017b) and incorporate a multi-tier rule-based reasoning logic.

Three agent types are accommodated in a continuous, 2D spatial environment with collision detection: Grass, sheep and wolf agents. These types form a primitive ecosystem with the grass being the most simple one. It is spawned at random positions in the environment based on a restricted-growth distribution function and serves as a food source for the sheep agents. Every sheep has a health value (game-style life points), an energy attribute and a hunger level. These properties express the sheep's condition and affect the actions it can undertake. The decision making happens in a four-staged rule system, where the available targets and the need for food are considered. Beside random movement (no hunger or no target perceived), the sheep can move towards a tuft of grass, eat it, and – if a certain energy level is satisfied and retained for a while – reproduce themselves, resulting in an additional sheep agent.

The above actions are implemented according to the *IODA (Interaction-Oriented Design of Agents* concept, developed by Kubera et al. (2011). By defining *interaction primitives*, the actions are not bound to specific agent types and can be re-used in a polymorphic fashion. This allows for the wolf agent to basically work in the same way. It lurks for sheep instead of grass by setting a different target filter, has slightly different energy values and an increased perception range. Apart from that, it is in terms of structure identical.

5.1.2 Parameterization

The table on the next page shows the default parameterization for the simulation model used throughout the experiments. Please note that several rows are marked with an asterisk: This means that the concrete value is subject to the specific experiment and may differ.

Model Parameter	**General**	**Grass**	**Sheep**	**Wolf**
Simulation steps	500			
Tick resolution (ΔT)	1 second			
Scale *	100.000 agents			
Environment *	500x500 continuous 2D			
Initial distribution		60%	25%	15%
Life points		60	80	100
Initial energy		2	50	70
Viewing range		-	8	10
Procreation min. energy		rnd.	60	80
Procreation duration		-	5	8

When using a carefully balanced parameterization (such as the values above), the *Wolves and Sheep* model gets into an almost steady state and shows up the characteristic predator-prey population fluctuations known from the *Lotka-Volterra* equations (as resumed by Itoh (1989)). Figure 5.1 gives an impression of the population dynamics:

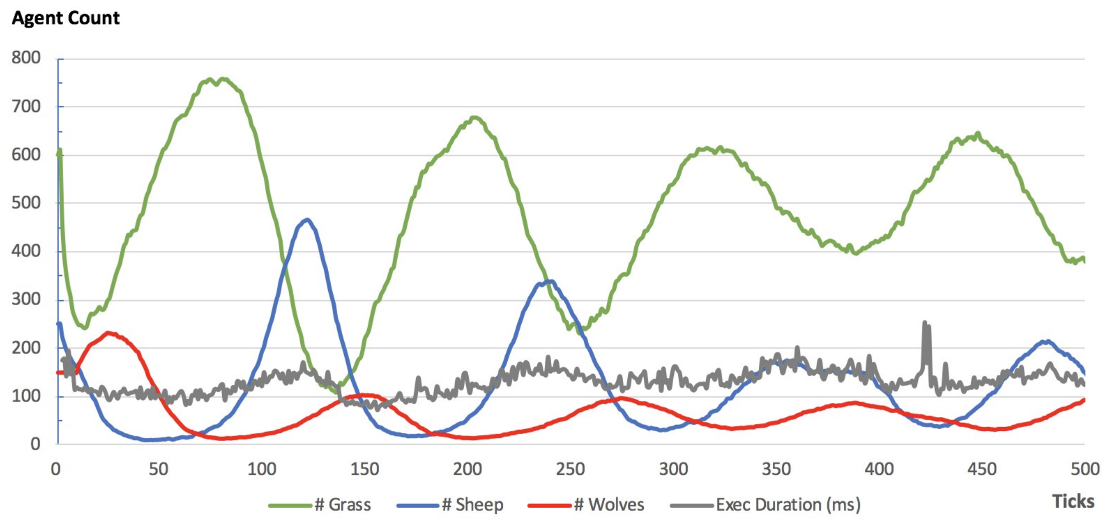


Figure 5.1: Population dynamics and execution time for a 1k run

Although the population densities are volatile, the total number of agents largely remains constant. For this reason it is expected to get comparable results despite the fluctuations. Furthermore, for the visualization it is of secondary importance whether a logically complex wolf agent or a simple grass agent is displayed. Obviously it is less costly to render a tuft of grass, but the same streaming, rendering and update routines apply. The 3D visualization uses the following presets for the experiments:

Visualization Parameter	**Value**
Field of view	6500
Query range *	50
Translation mode	cartesian
Elevation source	none (flat terrain)
Terrain preloading	8
Initial position	(25, -6, 5) (bottom left corner)

5.1.3 Simulation Configuration, Execution & Platform Specs

The reference model makes use of the *LIFE Components* library and is run with *LIFE Core v2.7.2*. This version offers the configurable result output as presented in section 4.2.1 with all features available. However, in this case a complete output is preferred to have all agent information as tooltip in the visualization available. In addition only key frames are used to eliminate the need for result preprocessing prior to delivery. Regarding the execution, two different platforms are used for the experiments. As a matter of course, a full-scale run has to be done on the Kubernetes cluster in order to prove integration, scaling capabilities and suitability for the MARS cloud. But unfortunately it is not possible to reserve computing power on the cluster, nor is it guaranteed to have the same preconditions for all experiments.

For that reason, the actual comparisons are done in a smaller scale on the local machine, a 2017 13-inch MacBook Pro with a 3.5 GHz *Intel Core i7* and 16 GB DDR3 RAM running Docker 18.06.0-ce. The databases are on an internal 500 GB PCI-Express flash storage and as for the 3D visualization, an *Intel Iris Plus Graphics 650* with 1536 MB RAM does its duty. To keep the resources under control, not the entire cloud setup but only the essential services are started. For the simulation and for experiments including live visualization, these are *full-pipeline setups* (consisting of databases, model analyzer, result configurator, simulation engine and visualization parts), whereas a pure visualization experiment only requires a *reduced setup*, containing of a pre-filled database and the visualization service plus browser frontend. Details are given in the concrete experiment descriptions underneath.

5.2 Scalability Experiments

The first number of experiments targets the visualization's suitability for large agent counts and further investigates the interplay of its settings and the resulting performance. For each of these experiments, three runs are conducted. The averaged results are presented in section 6.1.

S-1: Large-scale feasibility

The first experiment measures the influence of the total number of simulation entities on the overall visualization performance and responsiveness in order to prove hypothesis H2 to be true. To achieve comparability, it uses the *Wolves and Sheep* model executed on the local machine. Based on the default parameterization, three deviations regarding the total agent count and the environmental extents are made:

- **S-1.1:** 1000 agents, 50x50 environment
- **S-1.2:** 10.000 agents, 150x150 environment
- **S-1.3:** 100.000 agents, 500x500 environment (default)
- **S-1.4:** 1.000.000 agents, 1500x1500 environment

For every setup, the average frames-per-second (FPS) of the rendering engine are measured as well as the initialization time (time from HTTP loading request until first tick is visualized) and the responsiveness ("time-to-tick", the waiting time *tick query* → *tick display*). Because a very long simulation execution time has to be expected for the last two experiments, these are run upfront and only for 10 ticks. This means that during benchmarking time, only the reduced setup is active.

S-2: Impact of the query range

After the scale independence has hopefully proven true, the second experiment series aims at examining the impact of the query range on the performance and responsiveness. The simulation results of the 100k runs of S-1.3 are re-used in the local visualization setup and this time varying query ranges are applied:

- **S-3.1:** query range 25
- **S-3.2:** query range 50 (default)
- **S-3.3:** query range 100
- **S-3.4:** query range 150
- **S-3.5:** query range 200

For all experiments the averaged engine FPS, the network traffic required to download the visualization data and the time-to-tick as well as the number of entities visible on the screen are monitored.

S-3: Autoplay bandwidth requirements

The two experiments above used the manual playback for tick visualization. This third series investigates the bandwidth requirements for various autoplay settings as well as the processing load on both sides. This time the results of the 10k runs (experiment S-1.2) are visualized, because we want to try out even high TPS settings which might not be possible on the 100k run. Still, the same the visualization presets apply.

- **S-4.1:** 0 ticks / second (off)
- **S-4.2:** 2 ticks / second
- **S-4.3:** 5 ticks / second
- **S-4.4:** 10 ticks / second
- **S-4.5:** 15 ticks / second

S-4: Cluster integration and visualization over internet

Given that the previous experiments succeeded, the last trial is executed in the MARS cloud and shall demonstrate that the visualization is completely integrated and cluster-capable. It uses the same setup and measured values as the 100k run (S-1.3), with the difference that the simulation is executed in the Kubernetes cluster. The results are displayed live, using the visualization service deployed in the MARS infrastructure and the integrated frontend with all traffic being routed through the internet.

5.3 Proof of the Generality Claim

Hypothesis H1 claims that this visualization shall be of a general character and applicable to a variety of different simulation models. In order to prove this, it is attempted to visualize two selected models which differ considerably from the predator-prey scenario. Both models are developed and elaborated by the MARS Group in cooperation with external stakeholders and use geospatial real-world data.

5.3.1 Transfer to the *Smart Open Hamburg* Model

The first real-world showcase is the *Smart Open Hamburg* model, a traffic simulation developed by Weyl et al. (2018) of the MARS Group. As a part of the *ahoi.digital* initiative to promote local cross-university research projects in the field of computer science, this model is developed together with the Hamburg University (UHH). It features a random car driver model taking place in the streets of Hamburg. An arbitrary number of car agents can be instantiated in the vicinity of the Bramfeld district, located in the north-eastern part of the city. The street network is expressed as a graph stored in an ArangoDB and made available to the agents by means of a custom spatial graph environment. The cars move along these edges with a resolution of one second per tick which allows a real-time simulation and visualization.

At every intersection, a car chooses randomly a new road to continue its journey. Though this behavior produces no meaningful results, it generates street traffic and offers a good opportunity for 3D visualization of urban scenarios. A car possesses several static attributes, like its length, acceleration and the safety distance to adhere to the preceding vehicle. Dynamic output properties are the current speed, targeted speed, current traffic lane and of course the position, expressed as a (long, lat) GPS coordinate.

The traffic simulation is run online with the latest version of LIFE currently available (v2.7.2), all properties flagged for output and in a full-pipeline setup. 500 cars are spawned and simulated for 1000 ticks with the visualization switched on as soon as results become available. The car positions form the connecting link between the simulation results and auxiliary data, because they allow the streaming of real-world terrain information with the cars projected on top of it. The backend's terrain provisioning service delivers map tiles for the query area as presented in section 4.3.3 and renders them using street textures and satellite imagery of Hamburg.

5.3.2 Transfer to the Kruger National Park Model

A simulation model of the Kruger National Park in South Africa (in short: *KNP model*) serves as the second showcase for the visualization's versatility. It is developed as part of the joint research project ARS AfricaE which aims to investigate the resilience of Southern African savannah ecosystems to land use and climate changes in order to predict ecosystem dynamics and to develop sustainable management strategies. Together with three other German and six South African partner institutions, the HAW Hamburg participates in this project with MARS situated in the third work package responsible for data integration and IBM.

The KNP model features two agent types: *Elephants*, which live in herds and roam the park in search for food and water and *Marulas*, a dominant tree species serving as food and shade source for the elephants and also usable as biomass indicator. The elephant agents incorporate several movement patterns, herd affiliation and a nutrition and maturing lifecycle. They move, drink and eat the Marula fruits, through the latter they also spread the Marula trees to new regions. Real-world elephant data containing position, herd affiliation, sex and age were in advance collected, allowing to instantiate 7500 elephant agents from a CSV file. The Marulas, on the other hand, express a growing function, fruit generation and aging and also are initialized from a CSV file. By means of image recognition and random distribution, roughly 5 millions of Marula trees were generated and can be deployed to the simulation.

Apart from these two agent types, also manageable water holes were realized as agents which can be optionally integrated to allow the simulation of different water management strategies. KNP utilizes a multitude of data layers fed by source files to integrate environmental features: IPCC time-series data are used to depict different scenarios for precipitation and temperature changes. Shade and biomass distributions are available as GIS raster layers (ASC files), the borders of the national park and its rivers are expressed using GIS vector layers (SHP files). The total area of KNP spans almost 20.000 square kilometers, making it certainly a large-scale simulation.

For initialization, the KNP model requires the above-mentioned agent init CSVs, time-series and GIS files as well as a mapping in the WebUI, which assigns all these files to their respective simulation entities (agents and layers). A full scale run with all five millions of agents is attempted and due to the resource requirements it has to be executed and visualized online. 100 ticks are simulated with an hourly resolution, beginning at midnight of August 1st, 2010. Regarding the output and visualization configuration, all agent data are written out in every tick. The movement types (elephants: mobile, trees and water holes: stationary) are appropriately set, all static attributes flagged accordingly and the visualization config assigns fitting stock 3D models. It uses the default camera settings, real-world terrain and satellite imagery as overlay.

5.4 Added Value for Simulation Evaluation

In the course of the introduction it was multiple times emphasized that a 3D visualization should be particularly suitable for non-scientist users and shall offer an intuitive and simple-to-use result presentation tool. H3 claims this added value for the evaluation purpose and suggests a survey of different user groups to support this statement. This section elaborates the survey by defining the user groups, the tasks they shall carry out and the questions asked in order to get feedback.

5.4.1 Conduct of Survey

Three target groups are formed for the survey. These audiences shall represent different knowledge levels, fields of activity and objectives to be achieved with the visualization:

1. *Domain specialists:* The main focus lies on the MARS end users, mainly involved in building and evaluating simulation models. For that reason people from the application domain form the principal user group. Modelers of the KNP scenario and guest students were asked to evaluate the visualization for their needs and to give feedback of how useful it is for their purposes.

2. *MARS developers:* Next to the opinions of end users, also the appraisal of the MARS platform development team is of importance. They know the structure and realization of the entire MARS system and can direct their attention to technical features, such as extensibility, performance and maintainability during the survey.

3. *External people / bystanders:* As a third group, persons with no affiliation to MARS and the simulation business are surveyed. Unbiased by technical or domain aspects, this group will most likely rate the visualization on how intuitive the controls are, clarity of the user interface and visual quality and gives insights on the suitability for presentation purposes and usefulness for external stake holders and decision makers.

For each group, multiple individuals shall be asked to conduct the tasks mentioned below and to give feedback by answering a couple of questions during or after the evaluation. The survey is done in the online deployment and covers the following activities:

- Basics (movement and playback):
  - load the 3D visualization (the direct link for a publically available, pre-simulated *Wolves and Sheep* scenario is supplied)
  - change the perspective by moving the camera around
  - select an agent to retrieve additional information
  - use manual playback to display the next ticks
  - enable the autoplay feature and change playback speed
- Configuration:
  - open the result configuration dialogue for the model (link given)
  - change the 3D models
  - set a new initial camera position and change the skybox
  - perform a live-reload with the new settings
- Advanced (scenario switching and display settings):
  - use the top-down view camera lock
  - increase the query range
  - switch to another simulation (link for small-scale traffic scenario given)
  - set the camera to a specific real-world position
  - change the terrain texture overlay

The actual task description is embedded in a short tutorial and can be accessed as HTML document *(survey.html)* on the server.

5.4.2 Survey Criteria Definition

The user's satisfaction with the 3D visualization shall be made measurable by compiling a questionnaire. This form is divided into four sections and asks concrete questions to be answered by using rating scales. Five options are presented, ranging from "strongly disagree" over "neutral" to "strongly agree" and the questions intentionally vary between positive and negative statements to encourage a more thorough reading and consideration. In addition, free text fields are provided to allow the respondent to give own feedback.

- Usability:
  - *The user interface looks sleek and clean*
  - *I find the panels and buttons descriptive and had no problems following the tasks*
  - *The camera movement and interaction options are straightforward and intuitive*
  - *The visualization worked correctly on my device and in my browser*
  - *The external result and visualization config tool is easy to use*
  - *I would still prefer an integrated solution instead of a separate tool*
- Evaluation:
  - *The 3D scene makes it easy to understand the model's behavior*
  - *I used the agent tooltip to track the agents and their properties*
  - *The automatic playback mode is beneficial to validate movement pattern*
  - *[For modelers:] I would use the 3D visualization for my simulations*
- Visual quality:
  - *The visualization is appealing and pleasant to watch*
  - *Auxiliary features (terrain textures, skybox) help to create immersion*
  - *I think the lack of animations is a serious shortcoming*
  - *I would like an interpolation between the ticks during automatic playback*
  - *What about custom models? I want an uploader!*
- Performance:
  - *The visualization initialized fast*
  - *The rendering and camera movement was smooth*
  - *I experienced some breakdowns and had to reload everything*
  - *I consider the ineffective terrain loading algorithm a major showstopper*

The questionnaire was built using the online form builder JotForm and is embedded in the *survey.html* file. Section 6.3 of the next chapter presents the result of the survey, averaged over all participants in their respective group.

6 Results & Discussion

6.1 Scalability Evaluation

The first set of experiments dealt with the scaling capabilities of the 3D visualization and its backend. Section 5.2 presented a series of local and remote benchmarks regarding the implications of different agent counts, query ranges and autoplay settings. In order to get sound results, for each experiment three simulation runs and three visualization repetitions were made, resulting in nine measurements. This evaluation is based on the averaged data.

6.1.1 Local Test Series

Series S-1 examined the large-scale feasibility by running local simulation & visualization setups with up to 1 million of agents. The loading times (first start, reload, backend initialization and time to next tick) were tracked as well as the frames per second of the 3D engine.

Agent Count	Init Time	Reload Time	AMT Init	Time-to-Tick	FPS
1.000	1.721 ms	1.361 ms	360 ms	43 ms	60
10.000	3.281 ms	1.363 ms	1.918 ms	162 ms	58
100.000	3.662 ms	1.205 ms	2.457 ms	500 ms	58
1.000.000	26.567 ms	6.392 ms	20.175 ms	5.283 ms	57

As the above table and the data charts figure 6.1 on the next page show, the influence of the agent count on the visualization is very minor. There is always some basic effort of 1-2 seconds for initial loading needed (HTTP request, data transfer, JavaScript parsing etc) which shows up in the 1k initialization and the reloading times. For 10k and 100k, the initial loading times doubled, but though the agent counts for these runs deviate by factor 10, the loading times stay relatively close. The reloading times for 1k, 10k and 100k are almost identical, meaning that the frontend setup is mostly unaffected by the total agent count. The difference between initialization and reload is that in the former case, the backend has to create the agent meta table (AMT) for that session, while in the latter case it is only necessary to perform a recalculation of the camera's field of view and an alignment with the provided data.

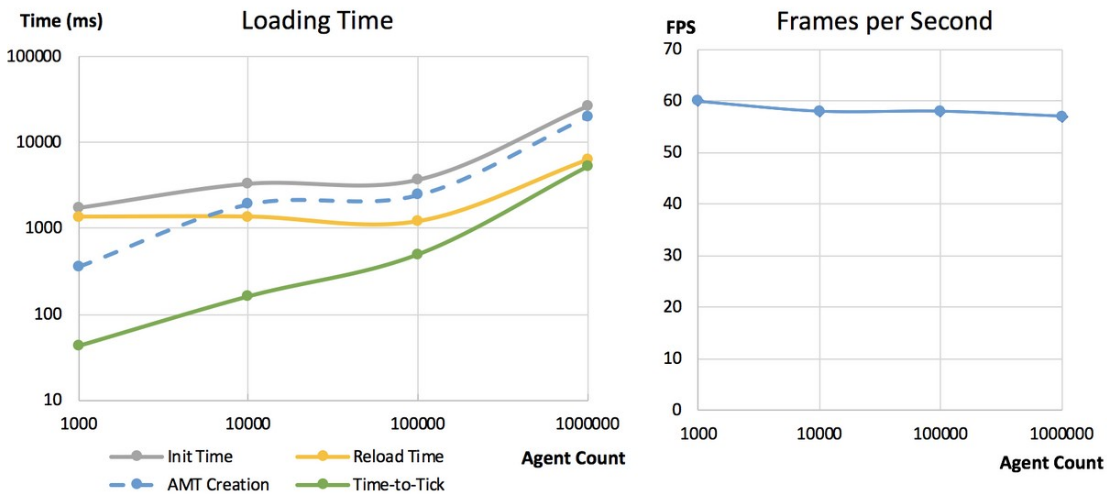


Figure 6.1: Impact of agent count on loading time and FPS

This gap between frontend and backend init time manifests in the last run with 1 million of agents: An increase in frontend init, reloading and backend init time was observed, ranging between factor 5 and factor 8 of the 100k run loading times. Considering that the agent count was raised by factor 10, this growth is of linear time complexity. The critical factor in that case is the AMT initialization, which takes 3/4 of the total loading time. If this could be optimized (see section 7.2), the initialization time may drop significantly. Notwithstanding, the higher the agent count, the more data have to be processed in the backend for the query range evaluation and for that reason, the reloading and tick-query times inevitably increase with the number of simulation entities. On frontend side, nothing of this backend preprocessing is witnessed (despite the increased waiting time) and the engine runs smoothly at around 60 frames per second all the time.

As next local experiment, different query ranges were evaluated. **Series S-2** measured the network traffic required for initialization (T_0) and next tick updates as well as the time until the requested tick is actually rendered. It also recorded the number of agents on screen and the engine performance. The data are presented as table and charts in figure 6.2 on the next page.

Both tick inquiries comprise a query for the complete area around the camera, whereby the initialization download includes both stationary and mobile agents. For further tick updates only the moving agents need to be considered, resulting in messages with approximately one third of the original size. A doubling of the query radius increases the packet size by factor 3 to 4 for both cases. Note that on camera movement additional messages would have to be sent, causing the update measurements to vary. The size of these additional messages depends on movement speed and direction but is definitively and significantly lower than the loading of the complete area, because only the new buffer zones have to be loaded. To avoid this trouble, the recordings for the above experiments were done with the fixed position stated in the setup.

The last experiment uses the eightfold of the query range compared to the first trial and the time-to-tick exhibits a restricted growth, not more than doubling itself. As was pointed out above, the time-to-tick is mainly caused by the backend-side intersection of the field-of-view. The same query routines apply, the range only affects the number of agents selected for retrieval and thus does not raise the response and processing time too much. It still increases the number of objects visible to the user exponentially, putting a heavy load on the 3D engine. As consequence, the frames-per-second drop down to less than 20 FPS for the last trial. This being the case, the FPS drop makes higher query ranges pointless.

Query Range	Init DL (T_0)	Update Tick	Time-to-Tick	Visible Objects	FPS
25	19 KB	6 KB	399 ms	388	60
50	66 KB	20 KB	480 ms	1.355	57
100	236 KB	70 KB	626 ms	4.817	36
150	507 KB	152 KB	778 ms	10.257	24
200	877 KB	263 KB	837 ms	17.648	19

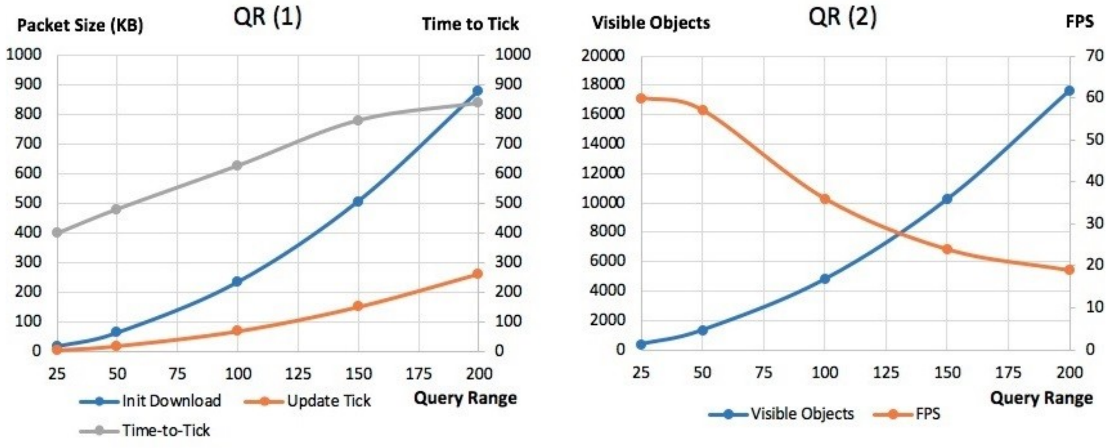


Figure 6.2: Query range effects comparison

After careful consideration the query range can be determined as the most important factor for rendering performance and network load. That being said, it depends on the average agent density which has to be determined separately for every simulation!

The last local test **series S-3** observed the impact of various autoplay settings to the resource demands of the visualization. Based on the 10k run, the loads for the client's CPU (packet processing), GPU (rendering performance) and the server load (result preprocessing) as well as the packet downstream were recorded. The results are presented and visualized below.

Ticks per Second	Frontend Load	Rendering Load	Server Load	Bandwidth
0	15 %	53 %	1 %	0 KB/s
2	17 %	55 %	2 %	26 KB/s
5	23 %	54 %	6 %	71 KB/s
10	27 %	57 %	10 %	138 KB/s
15	32 %	63 %	13 %	165 KB/s

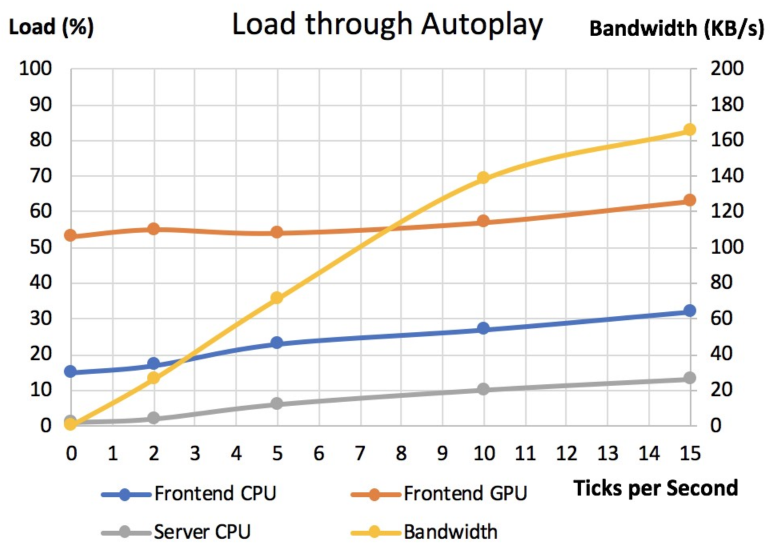


Figure 6.3: Processing and network load generated through autoplay

From the measurements it gets evident that an increase of the autoplay speed also craves for more resources. This behavior was expected, because just like for the manual playback, for each tick a range query on the entire data pool has to be made. With the server being responsible for this data pre-selection, it takes the main share of the load and its CPU demand rises the more frequently a tick is queried. The graph shows an almost linear increase from 1% when idle up to 13% at 15 TPS.

For the visualization client, every received tick requires to be incorporated into the 3D scene. The measured data above display almost the same increase for the CPU load, shifted by 15% base load (spent for user interface, network listeners and scene maintenance). Though the maximum load of 32% for 15 FPS seems to be much higher than the server load, the client's total increase is roughly by factor 2, whereas the server load had multiplied more than tenfold. Regarding the GPU load, an increase from 53% running idle up to 63% at 15 TPS could be noted, showing that the autoplay frequency only has a minor influence on the 3D rendering.

The most important measured value for the autoplay is the bandwidth requirement. With the majority of users presumed to be connected via internet, the available bandwidth specifies the upper limit for the TPS. For this 10k run with the default query range of 50, it showed a linear increase of ~13 KB per tick and second. The graph flattens out at the end, indicating that the time-to-tick was overshot and the actual tick rate would be more likely 12-13 TPS.

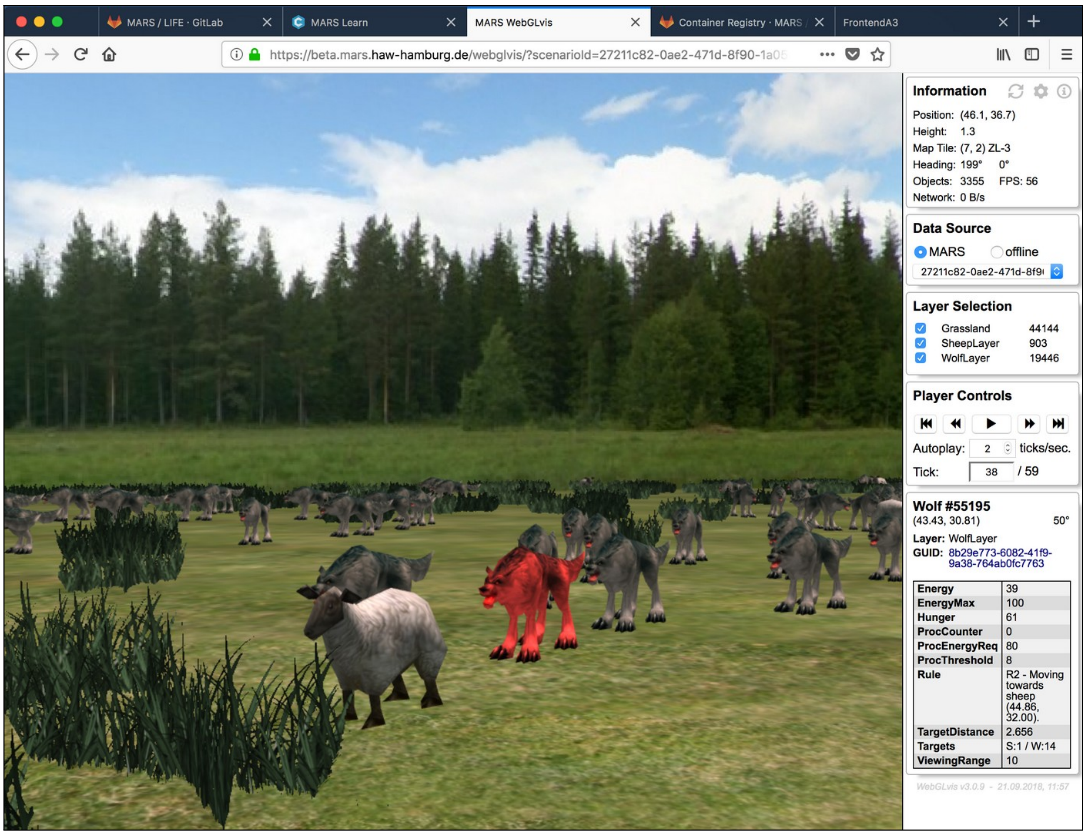
Recapitulating, it can be said that the autoplay setting plays a sensitive role for the frontend and backend performance. In order to prevent users from setting ridiculously high TPS values, the control element is restricted to 15 as the maximum value. Another self-regulating value is the latency ("time-to-tick") which impedes more frequent queries for large simulations or locally dense agent populations and thus automatically confines the load on server and client.

6.1.2 Cluster Integration

After the local scalability experiments delivered good results, it was then time to move to remote execution and to prove the cluster integration (experiment **S-4**). As stated in 5.2, a 100k version of *Wolves and Sheep* was uploaded, simulated with LIFE v2.7.2 and displayed with the VisualizationService in v3.0.9 (special version with benchmarking output enabled).

The experiment succeeded and is depicted in image 6.4 on the next page. The simulation run proceeded swiftly and the visualization initialized in about 2.7 seconds on average, which is 1/3 faster than the local run conducted during S-1.3. First, this seemed odd, because the visualization was run via internet and larger loading times were expected after all. The reduced loading time, however, was owned to the fact that the construction of the agent meta table takes a considerable amount of time, if it is not yet present (refer to the evaluation of *S-1*). With the visualization service running in the cloud infrastructure, it has more computing power to its disposal and gets this job done in about 60% of the time needed in the local setup.

The more potent host also allowed faster database queries and view range calculations, which dropped the time-to-tick by approximately 30%. Regarding the FPS, it has no effect whether the data originate from a local or remote source, which was as expected. But the reloading time was almost doubled (2.2 seconds), caused by the inevitably slower network connection. Though the university's network in general has more than enough bandwidth available and the *WebSocket-via-Ingress* configuration worked well, two connection failures arose and the link had to be re-established.


MARS / LIFE · GitLab
MARS Learn
MARS WebGLvis
Container Registry · MARS
FrontendA3
https://beta.mars.haw-hamburg.de/webglvis/?scenarioId=27211c82-0ae2-471d-8f90-1a05

Information

Position: (46.1, 36.7)
Height: 1.3
Map Tile: (7, 2) ZL-3
Heading: 199° 0°
Objects: 3355 FPS: 56
Network: 0 B/s

Data Source

[✓ MARS] offline
27211c82-0ae2-471d-8f9

Layer Selection

✓ Grassland	44144
✓ SheepLayer	903
✓ WolfLayer	19446

Player Controls

Autoplay: 2 ticks/sec.
Tick: 38 / 59

Wolf #55195
(43.43, 30.81) 50°
Layer: WolfLayer
GUID: 8b29e773-6082-41f9-9a38-764ab0fc7763

Energy	39
EnergyMax	100
Hunger	61
ProcCounter	0
ProcEnergyReq	80
ProcThreshold	8
Rule	R2 - Moving towards sheep (44.86, 32.00).
TargetDistance	2.656
Targets	S:1 / W:14
ViewingRange	10

WebGLvis v3.0.9 - 21.09.2018, 11:57

Figure 6.4: Execution of the *Wolves and Sheep* 100k run in the ICC cluster.
The wolf and sheep 3D models are property of *Blizzard Entertainment*.

6.1.3 Scalability End Result

Considering the results of the local test series and the online execution, it can be concluded that the 3D visualization has good scaling capabilities and it delivers a decent performance for the most scenarios. Of course the performance cannot be completely independent of the simulation scale, because at some stage a processing of the complete result space has to be done and this task inherently grows in expense the larger the data are. However, the visualization copes very well with high agent counts, has low bandwidth requirements and offers a convincing client performance on commodity hardware. Regarding the server-side, the cluster execution proved that more resources allow for a faster pre-processing and the outlook in section 7.3 provides additional hints on performance optimizations.

Due to the above restrictions, it shall be said that hypothesis H2 (scale/performance independence) is "largely" true.

6.2 Universality and Portability to Other Use Cases

The goal of the portability experiments was to prove the generality claim of hypothesis H1. To this end, two real-world models were executed and 3D-visualized.

6.2.1 Visualization of *Smart Open Hamburg*

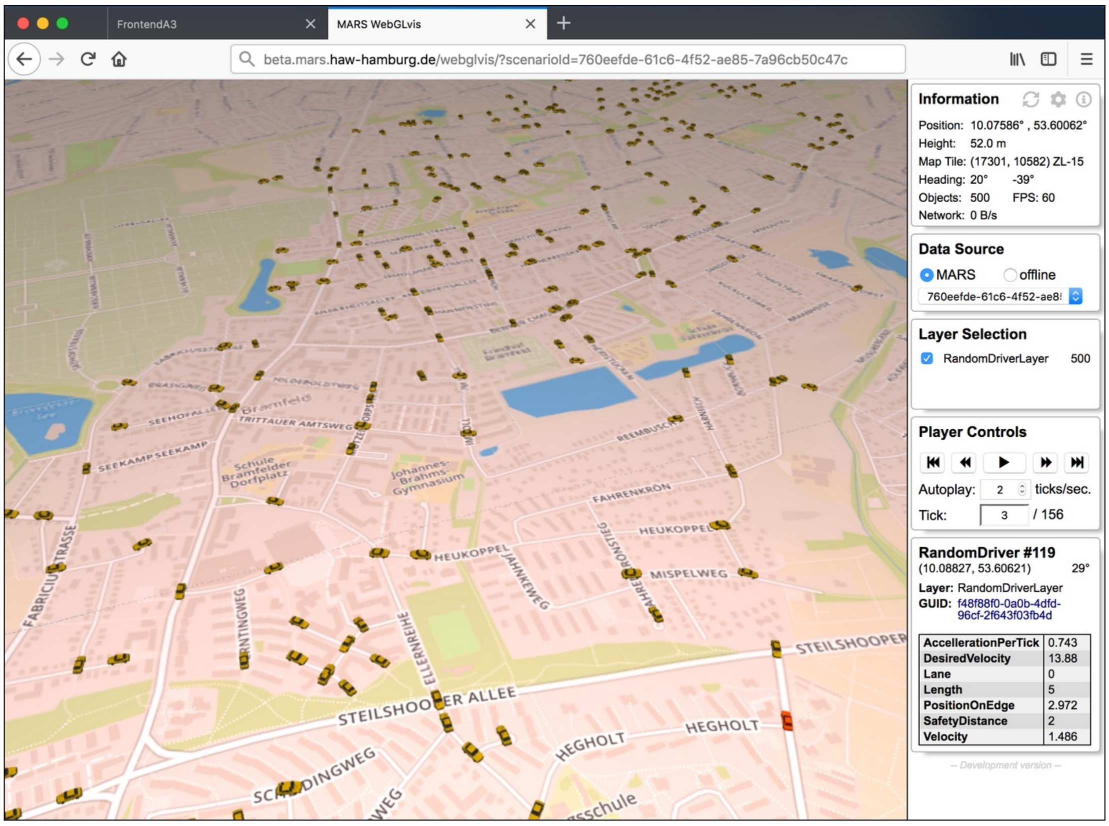
The first real-world showcase was the *Smart Open Hamburg* traffic simulation, described in experiment section 5.3.1. The required graph database was already running in the cluster and no model changes were needed, so the only thing left to do was to upload the model archive to the MARS cloud and to create the configurations. A total number of 500 car agents was set, running for 1000 ticks and with full output. As visualization setup, a yellow car 3D model was chosen to represent the car agents and the camera was preset to give a perspective view on the Bramfeld district.

Right after simulation start, the visualization was enabled. It loaded the cars at their initial positions very fast, although it took some time to fetch the surrounding terrain tiles. The camera movement was smooth and the engine ran constantly at a high frame rate, providing 55-60 FPS at any time. The time-to-tick stayed around 50 ms, allowing hypothetical autoplay settings of up to 20 ticks per second. With the camera staying at a fixed position or moving within close proximity to the loaded area, the overall performance was quite satisfactory.

One of the biggest problems perceived was the streaming of the real-world terrain (elevation and textures). The current algorithm loads a radius of map tiles for the given zoom level and has no transitions between these levels, e.g. to render tiles far away in a lower detail or to perform a smooth zoom in and out. This causes an unnecessarily high waiting time, in particular on zoom level change, which currently requires a complete reload. Especially when using satellite imagery, whose textures are about 5 times bigger in size, the loading times increase rapidly. Beside the large file size, satellite textures also made it more complicated to recognize the car agents, which is why the default street map overlay was chosen instead.

During the automatic playback, the visualization proved its value for debugging purposes. It showed up that the cars' orientation values were not properly updated when choosing a new road. This bug – probably hard to find when working with the plain data – became totally obvious by mere observation.

Screenshot 6.5 gives an impression of the visualization in action. The scene shows a view on Bramfeld at the intersection Bramfelder Chaussee / Steilshooper Allee in northern direction right after simulation setup (tick 3/1000). At the time of the screenshot, the simulation was still running and provided the visualization with a constant update stream.



Information
Position: 10.07586° , 53.60062°
Height: 52.0 m
Map Tile: (17301, 10582) ZL-15
Heading: 20° -39°
Objects: 500 FPS: 60
Network: 0 B/s

Data Source
[✓ MARS] offline
760eefde-61c6-4f52-ae8!

Layer Selection
✓ RandomDriverLayer 500

Player Controls
Autoplay: 2 ticks/sec.
Tick: 3 / 156

RandomDriver #119
(10.08827, 53.60621) 29°
Layer: RandomDriverLayer
GUID: f48f88f0-0a0b-4dfd-96cf-2f643f03fb4d

AccellerationPerTick	0.743
DesiredVelocity	13.88
Lane	0
Length	5
PositionOnEdge	2.972
SafetyDistance	2
Velocity	1.486

Figure 6.5: *Smart Open Hamburg* scenario visualized

6.2.2 Visualization of the Kruger National Park

The second showcase was the Kruger National Park (KNP) model which was described in section 5.3.2. Compared to the original model and experiment description, two small alterations were made: First, for the time-series data an in-memory time-series layer was used, which directly reads the CSV files for temperature, precipitation etc. and does not require an external *InfluxDB* instance. With this modification, the model was uploaded to the MARS platform and initialized as a large-scale run. This run, however, was aborted after a while in order to speed up the setup and execution time. Instead, a 12k run with all 7500 elephants and 5000 marula trees was performed.

The visualization was started as soon as the first few ticks were done and initialized in reasonable time. As default camera position an area east of the Skukuza Camp was chosen for observation. Several zoom levels, camera angles and view settings were tested and in general the 3D visualization came up to the expectations. Two takes are presented in figure 6.6 on the next page, one giving a map overview of the scenery and the other one featuring a ground-level view portraying an elephant in close-up.

A more thorough look at the scene from above also helped to discover some potential flaws in the model. As it gets visible from the upper screenshot, the reduced marula agent count did not lead to a more sparse yet even distribution, but rather resulted in one elongated patch with a very dense vegetation. Furthermore, the placement does not take any other terrain features into account, resulting in trees standing on the road and growing in the middle of a river. Regarding the elephants it seems that the herd affiliation does not work as expected, because when looking at groups of elephants standing closely together, the individuals most often have distinct herd IDs. For the latter observation it also turned out that an hourly resolution is not so suitable for the visualization in 3D because it allows the elephants to interact and move over large distances, which makes it hard to track an individual.

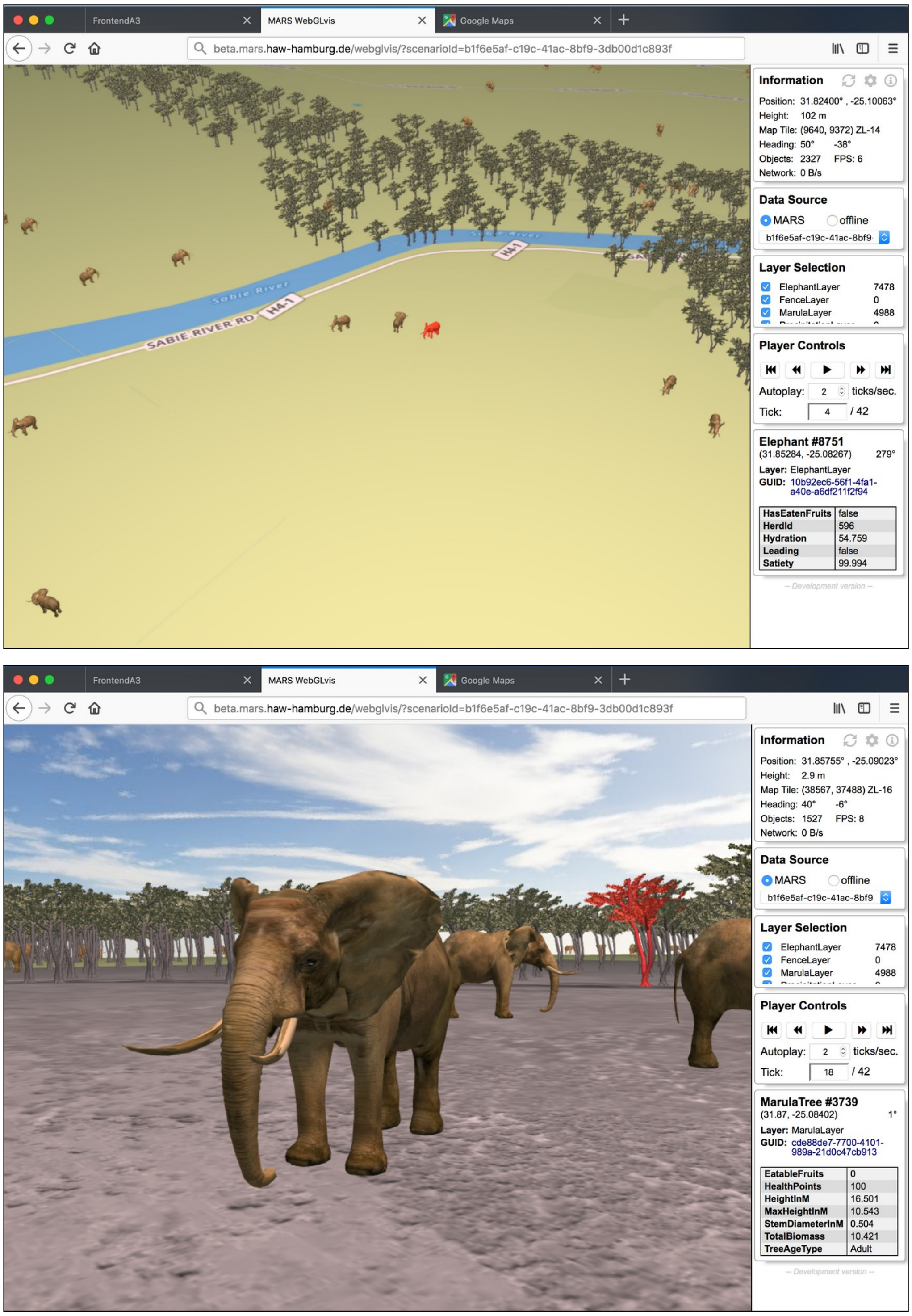
For the visualization itself, a few shortcomings could be noted. Right now, the marula trees have a huge impact on the rendering performance, because the current model is way too detailed (around 64k polygons) and another level-of-detail for far-range trees is urgently needed. As a remedy, an adjustment of the query range to 0.1 helped to cull a majority of those trees further away and gave an acceptable performance of 6-10 FPS for roughly 2000 entities to be rendered. When disabling the marula layer, the FPS rebounded to the usual 55-60.

Another problem was the terrain loading mechanism and the same problems as for the *Smart Open Hamburg* model occurred. In addition, it seems that for some tiles and zoom levels no elevation maps are available from the Mapbox API, resulting in zero-elevation tiles ruining the scenery. For that reason, the elevation on the two screenshots was disabled by setting the height source to "none / flat area" in the *ResultConfigService*.

6.2.3 Assessment of the Generality Claim and Usefulness

In summary, the 3D visualization worked very well for these two very disparate simulations. Despite the terrain streaming problems discussed above, the visualization proved to be capable of both cartesian and real-world geospatial scenarios, independent of the domain and simulation scale. When it comes to model evaluation and debugging, the 3D view turned out to be a beneficial and intuitive tool. And last but not least, the usage of the 3D visualization only requires a minimal configuration overhead from the user. For every agent type a model needs to be chosen, the initial camera position has to be set and that's it!

In conclusion, the hypotheses H1 (generality claim), H3 (evaluation/debugging value) and H4 (low configuration overhead) can be confirmed to be true.


FrontendA3
MARS WebGLvis
Google Maps
beta.mars.haw-hamburg.de/webglvis/?scenarioId=b1f6e5af-c19c-41ac-8bf9-3db00d1c893f
Sabie River
SABIE RIVER RD
H4-1
-- Development version --

Information
Position: 31.82400° , -25.10063°
Height: 102 m
Map Tile: (9640, 9372) ZL-14
Heading: 50° -38°
Objects: 2327 FPS: 6
Network: 0 B/s

Data Source
● MARS ○ offline
b1f6e5af-c19c-41ac-8bf9

Layer Selection
✓ ElephantLayer 7478
✓ FenceLayer 0
✓ MarulaLayer 4988

Player Controls
Autoplay: 2 ticks/sec.
Tick: 4 / 42

Elephant #8751
(31.85284, -25.08267) 279°
Layer: ElephantLayer
GUID: 10b92ec6-56f1-4fa1-a40e-a6df211f2f94

HasEatenFruits	false
HerdId	596
Hydration	54.759
Leading	false
Satiety	99.994

Information
Position: 31.85755° , -25.09023°
Height: 2.9 m
Map Tile: (38567, 37488) ZL-16
Heading: 40° -6°
Objects: 1527 FPS: 8
Network: 0 B/s

Data Source
● MARS ○ offline
b1f6e5af-c19c-41ac-8bf9

Layer Selection
✓ ElephantLayer 7478
✓ FenceLayer 0
✓ MarulaLayer 4988

Player Controls
Autoplay: 2 ticks/sec.
Tick: 18 / 42

MarulaTree #3739
(31.87, -25.08402) 1°
Layer: MarulaLayer
GUID: cde88de7-7700-4101-989a-21d0c47cb913

EatableFruits	0
HealthPoints	100
HeightInM	16.501
MaxHeightInM	10.543
StemDiameterInM	0.504
TotalBiomass	10.421
TreeAgeType	Adult

Figure 6.6: Map view on the Kruger National Park and elephant in close portray

6.3 External User Feedback

The user survey of section 5.4 was forwarded to roughly 35 persons across several channels. Unfortunately, the survey found little reaction and only four submissions were received. Looking back, the question arises whether the public interest was too low, the time requirement too high or if maybe the long description text scared away people right from the start. A statistical interpretation is senseless for four submissions, but nevertheless these results shall not be ignored because they offer a very positive feedback and valuable suggestions.

The survey started with usability questions and on the topic of ease-of-use and interface design the attitudes varied widely. This was just as expected – what seems to be pretty and intuitive for one person, may be functional for the other and counterintuitive for the third. However there were no negative answers, but largely positive votes and the visualization worked trouble-free on all systems (apart from some network problems, see below). Regarding the separate configuration tool, some contradictory views came up and the majority would favor an integrated solution, though at this point in time it is unclear how the overall pipeline may be altered to serve this task.

Outstanding feedback was received on the questions regarding the visualization's benefit for result evaluation. All participants stated that they would definitely employ this tool and extolled the knowledge gained by 3D rendering. The possibility for automatic playback enjoyed popularity and also the tooltip-based agent attribute explorer was met with a very positive response. After that, the solution's visual quality was rated. The overall reception ranged from good to very good, but regarding future extensions, opinions tend to differ sharply: While most respondents consider animation as a non-vital but helpful addition, one person was completely against it. In a similar way, one half does not care about the movement interpolation, while the other half judged it irritating. More consensus could be reached for the uploading tool for custom 3D models: Nobody thinks negative of it and for many this is a requested feature.

Though several bottlenecks exist, all participants were content with the current performance and loading times. The inefficient terrain loading algorithm was still tolerable for the most users and the engine and movement smoothness was satisfying. But many people suffered of network problems and for further development, the elimination of this cause of trouble should be given priority before additional features are implemented.

As last action, the surveyed users were asked to rate their overall level of satisfaction. For this rating, the visualization achieved a gratifying value of over 90%! Also the comment text field provided helpful insights into desired additions or changes, e.g. for other control options or new 3D models. In summary, the survey took a disproportionate effort but was beneficial.

7 Conclusion

With all the experiments conducted and evaluated, the time has come to draw a conclusion of what was done and achieved in this thesis. This chapter comments in retrospect on the goals defined at the outset and also presents some starting points for future work.

7.1 Achievements

At the beginning, this thesis made four assertions the visualization shall adhere to: First and foremost it claimed that it should be possible to design and build a model-independent visualization that works regardless of the simulation's scale and thus can cope with millions of agents situated in vast, real-world terrains. Of course, such a visualization is not an end in itself, but should rather serve as a valuable evaluation and demonstration tool for modelers and non-expert users alike, which requires a high degree of usability.

When looking at the results from chapter 6 and the subsequent discussion, it becomes clear that these goals were by and large reached. The portability experiments proved an independence of the domain and showed that the visualization handles artificial and real-world scenarios equally well and does not make a difference whether it is a predator-prey scenario, a traffic simulation or an ecology model. Also the scaling experiments had the desired outcome and confirmed that the client's query range around its current camera position is the pivotal factor for visualization load and performance, not the total agent count itself. However, the large-scale runs showed that there is no complete independence – as conclusively discussed in section 6.1.3, all data have to be processed which inevitably leads to longer loading and waiting times the larger the datasets are. The next chapter 7.2 addresses this shortcoming more detailed. Until then, the visualization only provides limited scale-independence.

That a 3D visualization is useful for simulation debugging and evaluation became apparent multiple times: For instance, the bugs found during the portability experiments manifested almost on their own by mere observation. Furthermore, the user survey returned a very positive feedback. Though the degree of participation was less than hoped and expected, the public opinions were across-the-board impressed and uplifting. Almost all participants praised the evaluation capabilities and the majority liked the user interface and affirmed its usability.

To sum it all up, the solution designed and built during this thesis satisfies most of the self-defined goals. As a concluding remark and as a personal perception, I am very content with the achieved result. Even though the current state of the visualization is still far from finished and the road to it was long and winding – this endeavor was challenging, instructive and loads of fun at the same time! I hope that this visualization does a good service for the MARS platform and in particular for all fellow students maintaining the project and all modelers building on MARS. The next two sections shall guide a way to future improvements and further additions.

7.2 Unresolved Issues

During experimentation and test evaluation a couple of issues came up that should be tackled in near future. The main culprit for the long loading times is the construction and maintenance of the agent meta table (AMT). For the current solution, this in-memory table is the key structure to manage the agent data and essential to avoid database queries for every single action the user performs. Still, for the initial creation all agent metadata have to be retrieved from the database and need to be processed, which adds to the waiting time for the first client connection (0.3s for 1.000 agents, 20s for 1.000.000 agents, refer to section 6.1).

Even worse, when live-watching a simulation, its AMT is not read-only but gets constantly updated. For every spawned agent an entry has to be added and for removed agents, the deletion flags have to be set. Ultimately, if these changes affect stationary agents, they have to be synchronized with all watching clients which are in viewing range. All these maintenance measures require a high degree of work, especially for fast-paced simulations or when doing this task for multiple simulations at once. As a consequence, the server load raises and the connection latency increases as well, but currently no more elaborate algorithm was devised.

Another problem with serious impact are the occasional connection failures which occur when the visualization service is deployed in the ICC cluster and accessed externally. These failures require a re-connection, but often the users don't realize what's wrong right away and first wonder why nothing works anymore. Without Kubernetes or when running in a local cluster no problems occur, so potential causes of failure may be the university network which is known to be liable to break down, the ICC cluster's internal networking or the Ingress configuration. To narrow down the options, one might run the services in another cluster or host it in the HAW network directly and a possible remedy would be to automatically detect connection failures on the client-side and to negotiate a new connection. Nonetheless it makes sense to first investigate the source of the connection problem.

7.3 Outlook

The current visualization solution is far from perfect, but rather a practical and purpose-oriented first draft. This section presents a number of ideas for usability and performance optimizations for the complete pipeline in general and for the 3D visualization in detail.

Pipeline Optimization & Simplification

- *Automated change behavior analysis:* One major downside of the current output system with the storage saving strategies of section 3.3.1 is the increased overhead for the user to flag the static agent attributes accordingly. This process is both laborious and error-prone and may be replaced by an automated mechanism. In order to determine the change behavior programmatically, the *ReflectionService* of section 4.1.1 needs a major overhaul and has to perform a tracking of all attribute assignments throughout the code in order to detect whether it is only written once or may be changed during runtime.
- *Improved service coordination:* The *ResultConfigService*, the *ResultAdapter* and the *VisualizationService* form the backbone for the 3D visualization and work closely together. Right now they use a ring-buffer messaging system to propagate change events among themselves. A more elaborate communication with interest management could help to reduce unnecessary communication and increases performance. For example, no simulation updates with agent details have to be sent if there is no client live-watching that simulation and therefore no AMT has to be maintained.
- *Custom heightmap loader:* Initially it was planned to provide a file loader for custom heightmaps. Unfortunately, such a loader could not be realized in the available time. Still, it would be considered beneficial to offer this opportunity to the user, because currently no terrain for simulations with custom coordinate reference can be used. Such a loader would be realized as a terrain provider plugin for the backend service and should offer support for common GIS formats (such as ASC) and maybe PNG or RAW image files.
- *3D model uploader:* The addition of new 3D models is currently a manual process, requiring a number of preparation steps and a fundamental understanding of 3D modeling. Tools like *Blender* and a custom converter have to be used and the generated model needs to be uploaded and added to the model index. This task is tedious and a model uploader tool which accepts OBJ models and automatically takes care of these conversion processes may be of great benefit.

3D Visualization Improvements

- *Professional game engine:* With the 3D engine being just one small building block in the entire system, it was deliberately decided to keep it as plain and simple as possible. This weighting of a quick "DIY" solution versus an external subsystem with complex API, update and build processes was detailed in section 4.4.2. Nonetheless it can be worthwhile to start a second attempt to incorporate e.g. Unity 3D, Unreal Engine 4 or Unigine to deliver a stunning and state-of-the-art visual experience.

- *Animation:* The usage of 3D animation to express an agent's actions may help the spectator to discover what happened in the simulation and gives the impression of a vivid world, resulting in a higher immersion. However, it also poses a serious implication for the model design, because the reification of (inter-)actions becomes obligatory and an action-animation mapping is required. On 3D model side, a basic set of animations for commons actions is needed, such as walk, run, idle, die, spawn, and maybe eat or drink. It remains questionable how to add custom animations for concrete simulation models. Fortunately, the majority of the surveyed users rated this as a rather unimportant feature.

- *Interpolation:* Owing to the fact that the simulation is executed in discrete time intervals, the 3D entities cannot move smoothly between the ticks but jump though the scene. This makes it more difficult to track the agents and also has a negative impact for presentation purposes. A possible remedy could be to use (linear) interpolation between the positions for tick transitions, though this adds massively to complexity and implies an agent movement not covered by the simulation. For that reason, a considerable part of the users doubt if this is useful.

- *Layer visualization:* In the current release, the layer visualizer is just a stub, providing no functionality. It has to be implemented on both client- and server side in order to transmit and render layer information, such as time-series values. Beforehand it needs to be elaborated how these data shall be visualized.

- *Touch screen control:* Currently, the 3D scene navigation only works with keyboard and mouse or touchpad input. With tablets and smartphones becoming omnipresent, the support for touchscreen devices would be a suitable extension. Technically this is an easy task, but prior to the realization a clever navigation concept has to be developed.

Bibliography

[Al-Zinati et al. 2013] Al-Zinati, Mohammad ; Araujo, Frederico ; Kuiper, Dane ; Valente, Junia ; Wenkstern, RZ: DIVAs 4.0: A Multi-agent Based Simulation Framework. In: *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications* IEEE Computer Society (Veranst.), 2013, S. 105–114

[Al-Zinati and Zalila-Wenkstern 2015] Al-Zinati, Mohammad ; Zalila-Wenkstern, Rym: MATISSE 2.0: A Large-scale Multi-agent Simulation System for Agent-based ITS. In: *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2015 IEEE/WIC/ACM International Conference on* Bd. 2 IEEE (Veranst.), 2015, S. 328–335

[Assarsson and Möller 2000] Assarsson, Ulf ; Möller, Tomas: Optimized View Frustum Culling Algorithms for Bounding Boxes. In: *Journal of Graphics Tools* 5 (2000), S. 9–22

[Balci 1997] Balci, Osman: Verification Validation and Accreditation of Simulation Models. In: *Proceedings of the 29th Conference on Winter Simulation.* Washington, DC, USA : IEEE Computer Society, 1997 (WSC '97), S. 135–141. – ISBN 0-7803-4278-X

[Banks and Chwif 2011] Banks, J. ; Chwif, L.: Warnings About Simulation. In: *Journal of Simulation* 5 (2011), Nr. 4, S. 279–291

[Bellifemine et al. 2007] Bellifemine, Fabio L. ; Caire, Giovanni ; Greenwood, Dominic: *Developing Multi-agent Systems with JADE.* Bd. 7. John Wiley & Sons, 2007

[Bijl and Boer 2011] Bijl, Jonatan L. ; Boer, Csaba A.: Advanced 3D Visualization for Simulation Using Game Technology. In: *Proceedings of the Winter Simulation Conference*, Winter Simulation Conference, 2011 (WSC '11), S. 2815–2826

[Bijl 2009] Bijl, Jonazan L.: *How Game Technology Can Be Used to Improve Simulations.* 2009

[Capterra Inc. 2018] Capterra Inc.: *Capterra.com.* https://www.capterra.com/simulation-software/compare/95940-107009/AnyLogic-vs-FlexSim. 2018. – Accessed: 2018-07-23

[Ceranowicz 1994] Ceranowicz, Andy: Modular Semi-automated Forces. In: *Proceedings of the 26th conference on Winter simulation* Society for Computer Simulation International (Veranst.), 1994, S. 755–761

[Dahmann et al. 1997] Dahmann, Judith S. ; Fujimoto, Richard M. ; Weatherly, Richard M.: The Department of Defense High Level Architecture. In: *Proceedings of the 29th conference on Winter simulation* IEEE Computer Society (Veranst.), 1997, S. 142–149

[Dalski 2017a] Dalski, Jan: *An Output and 3D Visualization Concept for the MSaaS System MARS (SpringSim 2017 Presentation).* 2017

[Dalski 2017b] Dalski, Jan: The LIFE BasicAgents Package – A Generic SRA Agent Framework for the MARS Platform. (2017)

[Davidsson 2001] Davidsson, Paul: Multi Agent Based Simulation: Beyond Social Simulation. In: *Multi-Agent-Based Simulation.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2001, S. 97–107. – ISBN 978-3-540-44561-6

[Drogoul et al. 2013] Drogoul, Alexis ; Amouroux, Edouard ; Caillou, Philippe ; Gaudou, Benoit ; Grignard, Arnaud ; Marilleau, Nicolas ; Taillandier, Patrick ; Vavasseur, Maroussia ; Vo, Duc-An ; Zucker, Jean-Daniel: GAMA: A Spatially Explicit, Multi-level, Agent-based Modeling and Simulation Platform. In: *International Conference on Practical Applications of Agents and Multi-Agent Systems* Springer (Veranst.), 2013, S. 271–274

[Drogoul et al. 1994] Drogoul, Alexis ; Ferber, Jacques ; Cambier, Cristophe: Multi-agent Simulation as a Tool for Analysing Emergent Processes in Societies. In: *Proceedings of Simulating Societies Symposium*, 1994, S. 49–62

[Falge et al. 2012] Falge, E ; Brümmer, C ; Mukwashi, K ; Schmullius, C ; Hüttich, C ; Odipo, V ; Scholes, RJ ; Mudau, A ; Midgley, G ; Hickler, T et al.: SPACES Project ARS AfricaE–Adaptive Resilience of Southern African ecosystems. In: *Coordinates* 31 (2012)

[Fowler and Lewis 2016] Fowler, Martin ; Lewis, James: *Microservices.* `http://martinfowler.com/articles/microservices.html`. 2016. – Accessed: 2016-11-30

[Franklin and Graesser 1997] Franklin, Stan ; Graesser, Art: Is It an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. In: *Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages.* London, UK : Springer-Verlag, 1997 (ECAI '96), S. 21–35. – ISBN 3-540-62507-0

[Gat and Bonnasso 1998] GAT, Erann ; BONNASSO, R P.: On Three-layer Architectures. In: *Artificial intelligence and mobile robots* 195 (1998), S. 210

[Glake 2018] GLAKE, Daniel: MARS DSL: Eine typisierte Sprache zur Modellierungkomplexer agentenbasierter Modelle. (2018)

[Green et al. 1997] GREEN, Shaw ; HURST, Leon ; NANGLE, Brenda ; CUNNINGHAM, Pádraig ; SOMERS, Fergal ; EVANS, Richard: Software Agents: A Review. In: *Department of Computer Science, Trinity College Dublin, Tech. Rep. TCS-CS-1997-06* (1997)

[Grimm 1999] GRIMM, Volker: Ten Years of Individual-Based Modelling in Ecology: What Have We Learned and What Could We Learn in the Future? In: *Ecological Modelling* 115 (1999), Nr. 2, S. 129 – 148. – ISSN 0304-3800

[Grimm and Railsback 2013] GRIMM, Volker ; RAILSBACK, Steven F.: *Individual-based Modeling and Ecology.* Princeton university press, 2013

[Grundmann 2018] GRUNDMANN, Lukas: Globale Sensitivitäts- und Unsicherheitsanalyse mit MARS. (2018)

[Hüning 2016] HÜNING, Christian: Analysis of Performance and Scalability of the Cloud-Based Multi-Agent System MARS. (2016)

[Hüning et al. 2016] HÜNING, Christian ; ADEBAHR, Mitja ; THIEL-CLEMEN, Thomas ; DALSKI, Jan ; LENFERS, Ulfia ; GRUNDMANN, Lukas: Modeling & Simulation as a Service with the Massive Multi-Agent System MARS. In: *Proceedings of the Agent-Directed Simulation Symposium.* San Diego, CA, USA : Society for Computer Simulation International, 2016 (ADS '16), S. 1:1–1:8. – ISBN 978-1-5108-2315-0

[Itoh 1989] ITOH, Yoshiaki: Lotka-Volterra Equations. In: *J. Appl. Prob* 27 (1989), S. 900

[Jennings 2000] JENNINGS, Nicholas R.: On Agent-based Software Engineering. In: *Artificial intelligence* 117 (2000), Nr. 2, S. 277–296

[Karsten 2018] KARSTEN, Lennart: Optimizing Geospatial Read-Performance inside a Multi-Agent Simulation System. (2018)

[Kincaid et al. 2003] KINCAID, J P. ; HAMILTON, Roger ; TARR, Ronald W. ; SANGANI, Harshal: Simulation in Education and Training. In: *Applied system simulation.* Springer, 2003, S. 437–456

[Kornhauser et al. 2007] KORNHAUSER, D. ; RAND, W. ; WILENSKY, U.: Visualization Tools for Agent-Based Modeling in Net-Logo. In: *Agent2007, Chicago, November* (2007), S. 15–17

[Kubera et al. 2011] KUBERA, Yoann ; MATHIEU, Philippe ; PICAULT, Sébastien: IODA: An Interaction-Oriented Approach for Multi-Agent Based Simulations. In: *Journal of Autonomous Agents and Multi-Agent Systems* 23 (2011), Nr. 3, S. 303–343

[Law 2008] LAW, Averill M.: How to Build Valid and Credible Simulation Models. In: *Simulation Conference, 2008. WSC 2008. Winter* IEEE (Veranst.), 2008, S. 39–47

[Le Gall 1991] LE GALL, Didier: MPEG: A Video Compression Standard for Multimedia Applications. In: *Commun. ACM* 34 (1991), April, Nr. 4, S. 46–58. – ISSN 0001-0782

[Luke et al. 2005] LUKE, Sean ; CIOFFI-REVILLA, Claudio ; PANAIT, Liviu ; SULLIVAN, Keith ; BALAN, Gabriel: Mason: A Multiagent Simulation Environment. In: *Simulation* 81 (2005), Nr. 7, S. 517–527

[Manojlovich et al. 2003] MANOJLOVICH, Joseph ; PRASITHSANGAREE, Phongsak ; HUGHES, Stephen ; CHEN, Jinlin ; LEWIS, Michael: UTSAF: A Multi-agent-based Framework for Supporting Military-based Distributed Interactive Simulations in 3D Virtual Environments. In: *Proceedings of the 35th conference on Winter simulation: driving innovation* Winter Simulation Conference (Veranst.), 2003, S. 960–968

[Masse 2011] MASSE, Mark: *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces.* " O'Reilly Media, Inc.", 2011

[Michael and Chen 2005] MICHAEL, David R. ; CHEN, Sandra L.: *Serious Games: Games That Educate, Train, and Inform.* Muska & Lipman/Premier-Trade, 2005. – ISBN 1592006221

[Niemeyer 2016] NIEMEYER, Daniela: Goal-Oriented Action Planning für die Simulationsplattform MARS. (2016)

[Padilla et al. 2014] PADILLA, Jose J. ; DIALLO, Saikou Y. ; BARRACO, Anthony ; LYNCH, Christopher J. ; KAVAK, Hamdi: Cloud-based Simulators: Making Simulations Accessible to Non-experts and Experts Alike. In: *Proceedings of the 2014 Winter Simulation Conference.* Piscataway, NJ, USA : IEEE Press, 2014 (WSC '14), S. 3630–3639

[Parunak et al. 1998] PARUNAK, H Van D. ; SAVIT, Robert ; RIOLO, Rick L.: Agent-Based Modeling vs. Equation-Based Modeling: A Case Study and Users Guide. In: *Multi-Agent Systems and Agent-Based Simulation.* Berlin, Heidelberg : Springer Berlin Heidelberg, 1998, S. 10–25. – ISBN 978-3-540-49246-7

[Robertson et al. 1993] Robertson, George G. ; Card, Stuart K. ; Mackinlay, Jack D.: Information Visualization using 3D Interactive Animation. In: *Communications of the ACM* 36 (1993), Nr. 4, S. 57–71

[Russell and Norvig 2003] Russell, Stuart J. ; Norvig, Peter: *Artificial Intelligence: A Modern Approach.* 2. Pearson Education, 2003. – ISBN 0137903952

[Sklar 2007] Sklar, Elizabeth: *NetLogo, a Multi-Agent Simulation Environment.* 2007

[Thiel-Clemen 2013] Thiel-Clemen, Thomas: Designing Good Individual-based Models in Ecology. In: Wittmann, J (Hrsg.) ; Müller, M (Hrsg.): *Simulation in Umwelt- und Geowissenschaften, Workshop Leipzig* GI (Veranst.), Shaker, 2013, S. 97–106

[Tory and Moller 2004] Tory, Melanie ; Moller, Torsten: Human Factors in Visualization Research. In: *IEEE transactions on visualization and computer graphics* 10 (2004), Nr. 1, S. 72–84

[Vernon-Bido et al. 2015] Vernon-Bido, Daniele ; Collins, Andrew ; Sokolowski, John: Effective Visualization in Modeling & Simulation. In: *Proceedings of the 48th Annual Simulation Symposium.* San Diego, CA, USA : Society for Computer Simulation International, 2015 (ANSS '15), S. 33–40. – ISBN 978-1-5108-0099-1

[Vigueras et al. 2013] Vigueras, Guillermo ; Orduña, Juan M. ; Lozano, Miguel ; Jégou, Yvon: A Scalable Multiagent System Architecture for Interactive Applications. In: *Science of Computer Programming* 78 (2013), Nr. 6, S. 715–724

[Weyl et al. 2018] Weyl, Julius ; Glake, Daniel ; Clemen, Thomas: Agent-based Traffic Simulation at City Scale with MARS. In: *2018 Spring Simulation Multiconference*, 2018, S. 0–0

[Wilensky 1999] Wilensky, Uri: *NetLogo. Evanston, IL: center for connected learning and computer-based modeling, Northwestern University.* 1999

[Wooldridge 2002] Wooldridge, Michael: Intelligent Agents: The Key Concepts. In: *Multi-Agent Systems and Applications II.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2002, S. 3–43. – ISBN 978-3-540-45982-8

[Yamamoto et al. 2008] Yamamoto, Gaku ; Tai, Hideki ; Mizuta, Hideyuki: A Platform for Massive Agent-based Simulation and its Evaluation. In: *International Conference on Autonomous Agents and Multiagent Systems* Springer (Veranst.), 2008, S. 1–12

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 5. Oktober 2018 Jan Dalski