Master Thesis

Lennart Karsten

A Generic GIS-Based Layer Approach for the Multi-Agent Simulation Framework MARS

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Lennart Karsten

A Generic GIS-Based Layer Approach for the Multi-Agent Simulation Framework MARS

Master Thesis eingereicht im Rahmen der Masterprüfung

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

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

Eingereicht am: July 26, 2018

Lennart Karsten

Thema der Arbeit
A Generic GIS-Based Layer Approach for the
Multi-Agent Simulation Framework MARS

Stichworte
Geografische Informationssysteme, GIS, Raum-/ und Zeitbezogene Daten, Multi-Agenten Simulationen, Performance Analyse.

Kurzzusammenfassung
Diese Arbeit erweitert das Multi-Agent Simulationssystem MARS um einen generische GIS Unterstützung. Diese erlaubt es, die meist verbreitetsten Geo-Daten Formate in einer Simulation zu nutzen, ohne diese im Vorfeld konvertieren zu müssen.

Lennart Karsten

Title of the paper
A Generic GIS-Based Layer Approach for the Multi-Agent Simulation Framework MARS

Keywords
Geographic information system, GIS, spatio-temporal data, Multi-Agent Simulations, performance analysis.

Abstract
This work enhances the Multi-Agent simulation framework MARS, by introducing generic GIS support. This allows the use of the most common geo-spatial file formats inside a simulation, without complex file conversion.

Contents

List of Figures

Listings

1. Introduction

Agent-based modeling is a method of simulating complex environments from the prospective of an individual (Wooldridge 2009). Each, so called agent has parameters and actions, which define it's state. The agents interact with each other and their underlying environment to reach the next state. This approach allows to scale the number of agents, depending on the individual use-case.
Viewing each agent separately requires detailed information about it's behavior. This data includes the current position at a certain point in time, which makes it a good fit with Geographic Information Systems (GIS).

GIS is the process of storing, processing and visualization geographical data with reference in time and space (Maguire 1991). It is being used in several recent publications. Among them are city planing (Giles-Corti et al. 2016), remote sensing (Ghorbani Nejad et al. 2017; Pinto et al. 2017), bioclimatic comfort (Cetin et al. 2018) and simulations (Wang and Zlatanova 2016).
GIS provides the capabilities to handle complex geo-spatial operations, which are of great value to agent-based Systems.

Multi-Agent Research & Simulation (MARS) is the name of a working and research group at the HAW (University of applied Sciences) in Hamburg, Germany. It conducts in research on multi-agent simulations and develops its own simulation platform (Hüning, Wilmans, et al. 2014).
Most of the input data used to initialize models for MARS rely on GIS data. However, this data has to be converted to custom file formats before it can be uses.

This work focuses on introducing native GIS support to the MARS system. This increases versatility, by enabling the direct use of standardized GIS formats. The generic implementation allows to cover additional modeling use-cases without introducing new data layers. The design and implementation is done with the performance of the old system in mind, which ensures that the overall performance is not sacrificed.

The following section discusses hypotheses which will be examined during the process, to ensure the statements above.

1.1. Hypotheses

The MARS simulation system is a powerful framework that allows large-scale simulations and covers a wide variety of models. Adding GIS functionality is meant to improve the versatility. It's following hypotheses will be used to validate the changes which are being introduced.

H1 GIS data can be made usable in the simulation system, while retaining all functionalities and allowing to fulfill additional use-case.

The layers currently in existence inside the LIFE simulation system have a very narrow usage, where every layer covers a specific use-cases:

- A grid-based „Obstacle Layer“ that is capable of detecting obstacles between two points.
- The „Grid potential Field Layer“ manages a pre-generated raster that allows agents to sense a nearby location (e.g. water sources for animals).
- The „Geo potential Field Layer“ has the same functionality, but uses Global Positioning System (GPS) coordinates instead of a grid.
- The „TimeSeries Layer“ can store a single value for several timestamps.

These specific layers can be consolidated and replaced by a generic solution, that is able to be used for new use-cases.

H2 The import types can be replaced by one generic importer.

Each layer has it's own data type that has to be selected during import. This can be optimized to automatically select the proper type. Doing so reduces potential errors the users can make.

H3 It is possible to remove the custom file generators by relying on standardized GIS file formats.

Currently different types of data can be used inside a simulation. Each type needs a dedicated generator that converts data into the correct format. Relying on standardized GIS formats allows to make those converters obsolete.
Due to the environmental focus of MARS, most model developers, are already familiar with GIS. This allows to easily create, view and edit a given file with software, existing in the market today, like QGIS or ArcMap.

H4 The use of one generic GIS import and two generic GIS layers does not diminish simulation performance by more then 10%.

Massive simulations, for whom the MARS system was designed for, can contain of up to several million agent, with simulation times covering many years. Such runs can take weeks to complete. Therefore it is important that the changes introduced by the GIS layers don't reduce the performance of the overall system.
The generic structure of these layers have a certain disadvantage, compared to files that have been explicitly created for one thing. Therefore performance tests will have to be created, for comparison to the old system.

1.2. Motivation

This work focuses on integrating GIS capabilities into MARS. This increases the versatility of the system and makes it more appealing for a wider audience. This is being achieved by reducing the amount of tools and components involved in creating simulations. The use of standardized input types makes it easier to bring existing data into the system.
This approach also reduces complexity for the developers by decreasing the number of services that have to be updated and maintained. As a result, the number of potential fault locations is also minimized which allows the team to focus on other things.

My personal motivation for choosing the topic is the interest in GIS and the technical details of the system. Replacing a complex and error prone workflow with a generic solution, is a worthy challenge.
The topic also touches a wide area of the system. During the design and realization of the project, most parts of the systems were involved in some way. This made the project both challenging and interesting.

1.3. Structure Outline

The thesis is divided into several main parts. Chapter 2 provides the reader with fundamental knowledge on which this work is built upon. It covers the scientific background, as well as the technical basis. The analysis in Chapter 3 covers use-cases that the resulting product has to fulfill and the scope of this work. Chapter 4 is about the overall design of the software as well as the methods used. It is followed by Chapter 5 that covers the realization of the design. The experiments in Chapter 6 benchmark the use-cases specified previously. This is done by small deterministic models that cover each use-case and evaluate it, in comparison to the previous layers. This thesis ends with a summary, including the hypotheses validation and an outlook into future work.

2. Related Work

This chapter covers fundamental concepts, techniques ideas and technologies in the geospatial research area, as well as an overview of the MARS system. Section 2.1 covers different type of spatial data types, fundamentals of Coordinate Reference System (CRS) and renown technologies in the environmental space. Section 2.3 is about the MARS simulation system and its infrastructure.

2.1. Geographic Information System (GIS)

In the past years, mobile devices have revolutionized the way we work and interact with our environment. These mobile devices, like GPS, microphone, camera, accelerometer and many more, are equipped with sensors, which allow them to collect data in various ways. These modern devices are always connected to the Internet and transmit information about their location and other data to remote services.
This leads to vastly increased amount of data that is being gathered. Since Raw data is of little use, the value is generated by analyzing, categorizing and making the data able to be searched. The research involved in improving such techniques relies on GIS to a large extend.
GIS is the combination of all techniques that help store, analyze, manipulate and visualize geo-referenced data on earth.

2.1.1. Geo References and Projection

Standard imaging formats display data on a plane. This is the case for basically any picture or diagram, since it was created to be displayed on a screen or paper. That kind of data does not usually have a reference to specific locations. Exceptions like Geo-tagged images exist.
GIS data on the other hand is geo-referenced. Stored information does not only have a size, but every datum has a reference on a specific position on Earth. For this to work, the arc of Earth has to be taken into consideration.
Depending on the location and scale of data, different solutions have to be applied. When displaying at a small scale, like 1:200 for a city the arc of earth is a minor problem that one

might decide to ignore. On larger scales that involve opposed continents it is simply impossible to display them on a 2-dimensional surface like a paper or screen without distortion.

Map Projection Classes

Depending on the area being displayed, different types of distortions produce more accurate results. The most typical shapes are cylindrical, conical and planar projections as shown in figure 2.1.

Cylindric projection are best at the Equator, where depending on the implementation, there is minimal to no distortion. The accuracy decreases to a maximum at the poles, where the distortion becomes infinite and therefore is not usable anymore. This distortion makes areas at the poles appear far bigger than they really are. Greenland (2.34 million km^2) appears to be as big as Africa (30.37 million km^2), while in fact it is $1/13th$ of the size.
Fortunately for normal mapping purposes these thinly inhabited areas at the poles are not of a mayor interest. Therefore this type of projection is best for general usage.

When displaying data at the poles a planar projection produces the best results. It visualizes the desired pole the highest accuracy and decreases towards the equator. This type of projection cannot display the whole world in one image as seen in the corresponding graphic letter „c“. This makes the method unusable for general purposes. As a result, it should only be used for showing the poles.

The conical projection shown in the middle of figure 2.1 obtains the best results, when displaying either the Northern or Southern Hemisphere around ±45° latitude. When focusing on Europe, areas like Greenland seem smaller than they are and everything south of 45° latitude seams larger than it is. This results in everything below -45° latitude to not be usable anymore.

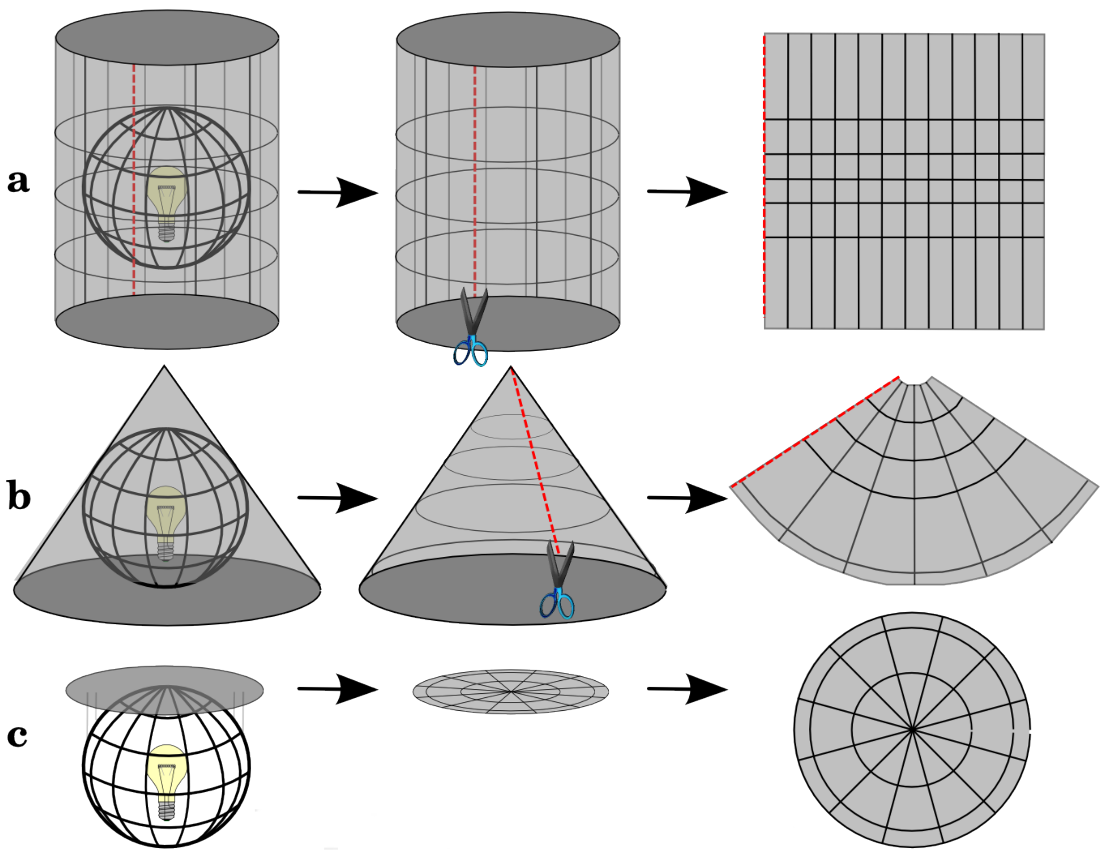

Figure 2.1.: The three most common shapes to project the Earth to. Source: qgis.org (2018)

Mercator Projection

Mercator is a cylindric projection with all the advantages and disadvantages mentioned in section 2.1.1. It is named after it's founder Gerardus Mercator, who first created the projection in 1569 (Meer 2012).
It quickly became popular because of its rhumb line course capabilities. This means that a straight line on the map is an actual straight course which was useful for nautical purposes. Also, the shortest course between two points on Earth is the direct straight line on the map.

The original map from 1569, as seen in figure 2.2, was already quite accurate for Central Europe, but had major flaws on continents further away. Over time, it has been optimized in accuracy. Today, it is the most used projection for general use, because it provides a good compromise overall. Figure 2.3 shows the modern Mercator map from 82° latitude to -82° latitude.

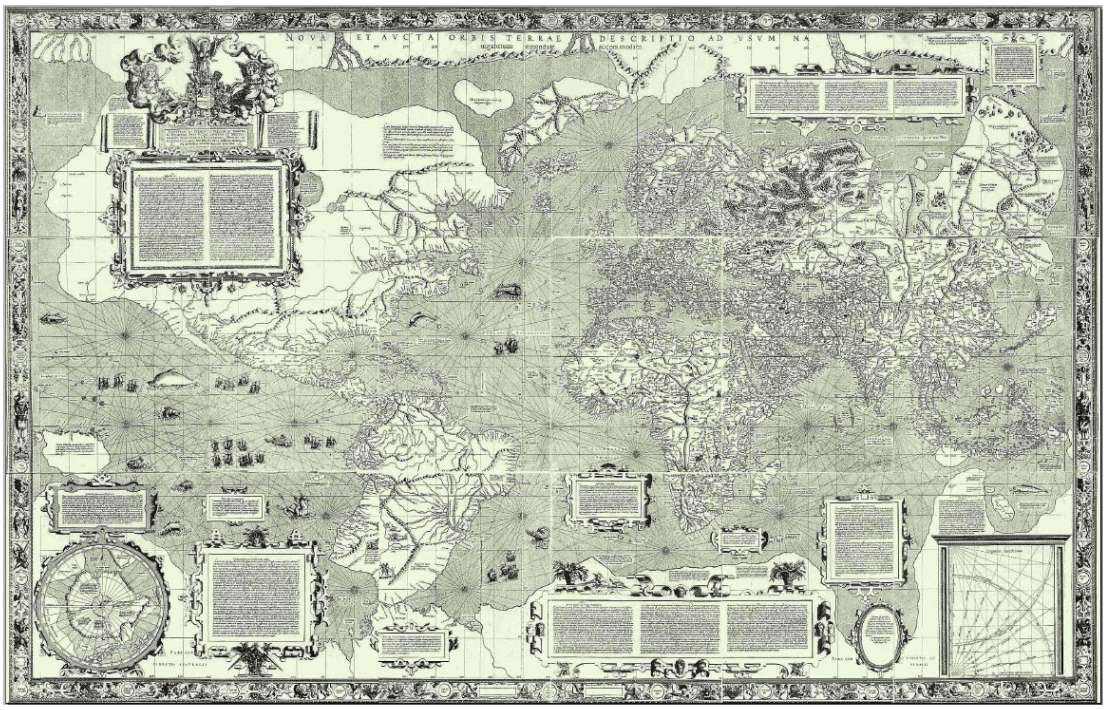

Figure 2.2.: Original Mercator projection from 1569. Source: Meer (2012)

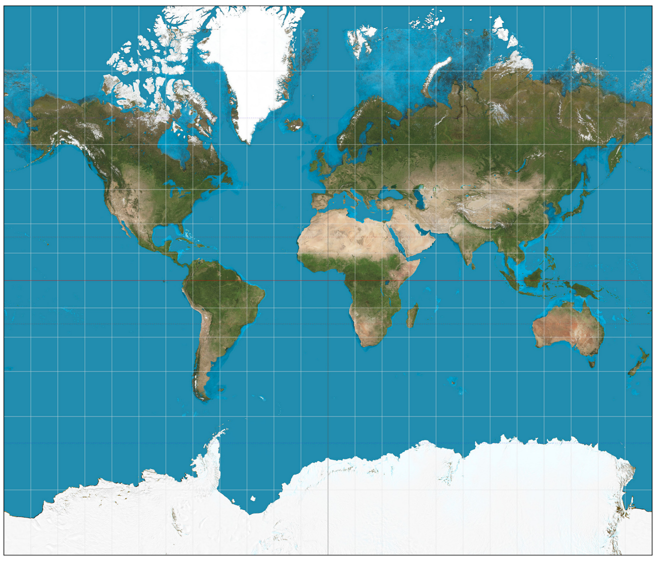

Figure 2.3.: Mercator projection of the world between 82° S and 82° N. Source: Strebe (2011)

WGS 84 (EPSG:4326)

The World Geodetic System 1984 (WGS 84) is an ellipsoidal coordinate system used for navigational purposes created by Decker (1986). It was first created in the $20th$ century, but has been improved in accuracy over time. 84 refers to the year 1984, when the latest revision of the WGS system was published. This version was improved with higher-fidelity data from the Earth Gravitational Model 2008 (EGM2008) (Pavlis et al. 2012). WGS 84 is also known as an European Petroleum Survey Group (EPSG) standard with the name EPSG:4236.
The coordinates origin in the center of Earth and are measured in degree latitude and longitude. The map on figure 2.4 shows the coordinates discussed in the following paragraphs.
Latitude has its 0° origin, also called prime meridian, in Greenwich England and increases east to a maximum of 180°, as well as west to -180°. Each vertical line connects the two poles. The maximum distance for one degree Latitude is 111 km at the Equator, decreasing to zero at the poles. Each meridian has a length of 20,003.93 km (half the Earth's perimeter).
Longitude has its 0° on the equator, and increases to the North Pole with a maximum of 90° and a minimum of -90° on the South Pole. One degree longitude is approximately 111 km at any point. The length of an Antimeridian varies from 20,003.93 km at the Equator to 0 on the poles.

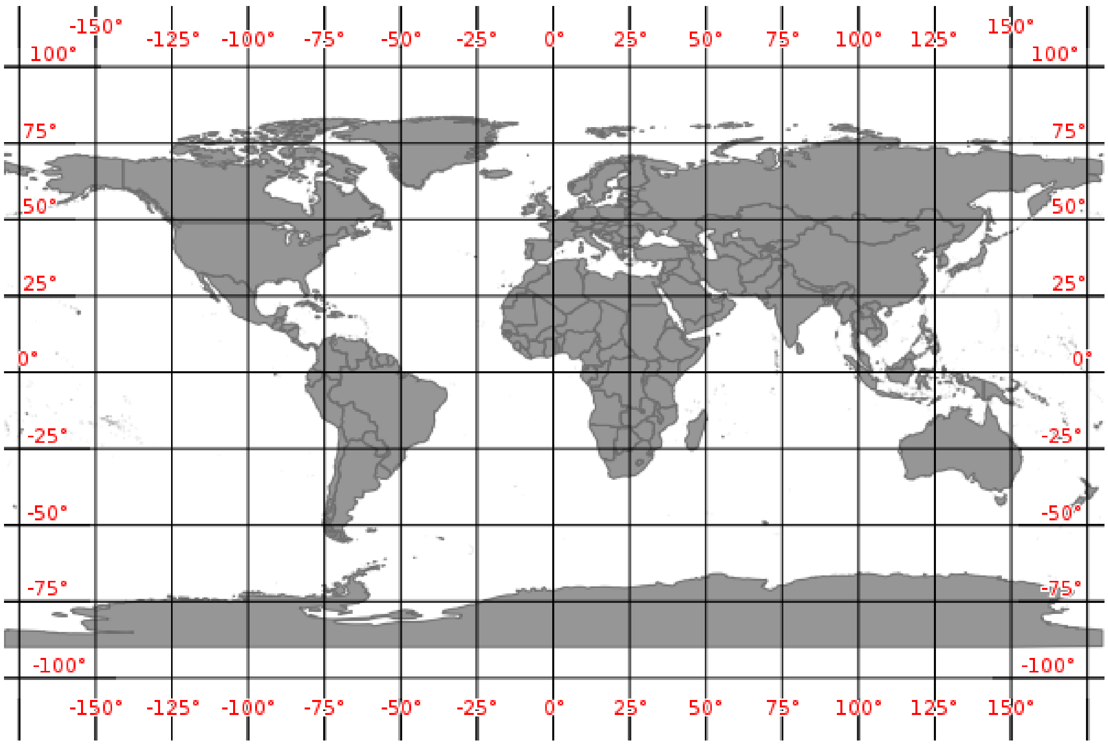

Figure 2.4.: The degrees of WGS 84 on a World map. Source: mapserver.org (2018)

WGS 84 / Web Mercator (EPSG:3857)

Web Mercator is an coordinate system based on WGS 84. While EPSG:4326 uses an ellipsoidal representation, EPSG:3857 is a projected, spherical version. It was created by Grafarend (1995). In difference to WGS 84 it uses Cartesian coordinates that are measured in meters. Web Mercator does not cover the whole Earth, but only ±85.06° latitude. This cuts off the poles but allows a square projection with a range of ±20,026,376.39m on both axis. The projection is not considered a geodetic system due to the missing 10 degree and its projection inaccuracies (National Geospatial Intelligence Agency 2014).

The Web Mercator projection was created for displaying the world on a screen in the web. The square shape allows the creation of so called Mercator Tile pyramids. This transfers the map data into tiles of identical size, like 265x265 pixel. On level 0 there is one image to show the world. On each level the number of tiles increases by a factor of 4 compared to the previous level. Figure 2.5 shows an example of such a Mercator pyramid.
This allows to load the required number of tiles in the desired level of detail without transferring gigabytes of data to the user. The mentioned approach was first used in Google Maps and is used by all renown map providers like OpenStreetMap, Microsoft's Bing Maps and Esri.

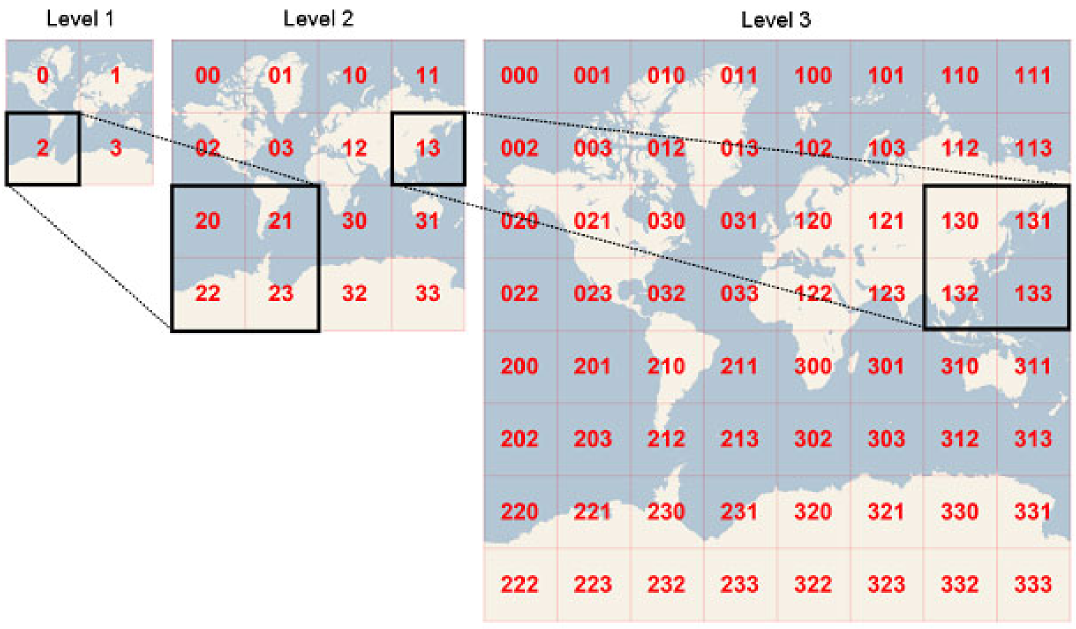

Figure 2.5.: A Mercator Tile Pyramid. Source: Schwartz (2018)

Although the projected version of WGS 84 is widely used, it is also criticized by GIS experts for its lag of accuracy. The projection not only sacrifices the poles, but it is also distorts the original projection. Web Mercator uses mathematical formulas and parameters that make it incompatible to WGS 84 (National Geospatial Intelligence Agency 2014). The errors increase with a larger distance from the Equator and can reach an offset of up to 40,000 meters. Figure 2.6 shows an overlay map of the United Kingdom, where it becomes most visible how severe differences are. The South of the UK shows an offset of 33,000 meters, while the North shows 36,700 meters.
For the reasons mentioned above, Web Mercator should only be used to visualize spatial data to the user, but never as a reference system for relevant calculations.

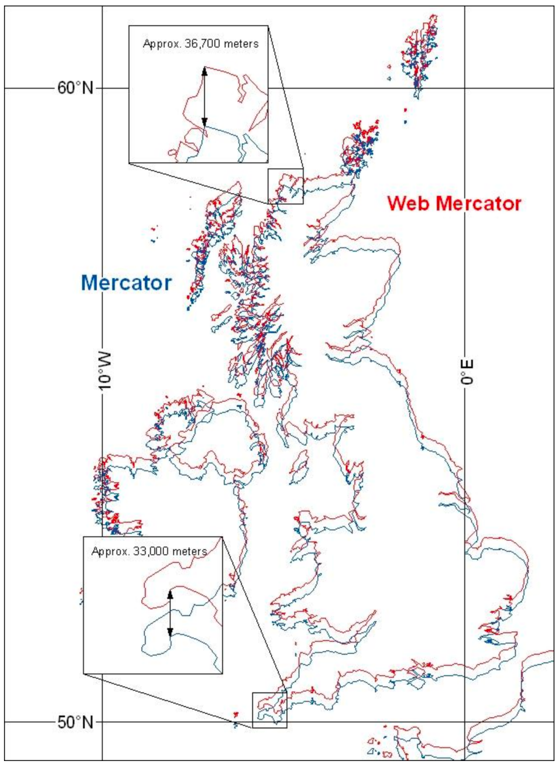

Figure 2.6.: Visual Example of Overlaying Ellipsoid Mercator and Web Mercator. Source: National Geospatial Intelligence Agency (2014)

2.1.2. Spatial Data Sources

GIS data is always geo-referenced, meaning each dataset has a location, and ideally a point in time, where it is valid. Nowadays, such data is almost always digital and since it has to be persisted in an efficient way, there are two ways GIS data is represented. These methods differ in their capabilities, accuracy and storage size.

Raster Data

Raster data represents information in a grid of, either square or rectangular shape with identical cell size. The grid cells are called pixels. Each pixel has a numeric value. 8 bit pixel depth is a common size for greyscale images, allowing a range of 256 values. Color is represented in 3 bands, each with a 8 bit value, leading to a 24 bit color depth. The bands are the red, green and blue (RGB) values.
GIS formats like GeoTIFF support a larger number of bands, since they are not pure image formats but are meant for storing data. A raster file could have two bands, one storing an elevation map in meters and another containing humidity values in percent.

Raster images are easy to understand and fast to read and write from, since no complex algorithms are involved. It is best for storing data with undefined structure in it. This includes images, and anything with a gradient of shapes and colors.
However, due to the nature of a grid system, storage consumption is not very efficient for certain types of information. E.g. when storing the position of values in a bitmap style, with 0 being no value and 1 being the location of a building. The grid size would have to cover the whole area and the cell size determines the accuracy of the position. This increases the file size, which is why raster data can become very big. Another disadvantage is the fact that the resolution is fixed.
When creating a file, a compromise between level of detail and file size has to be made. This cannot be changed afterwards. Figure 2.7 shows a shape, that is displayed in raster. Depending on the chosen number of pixels, the image is either fine (left) or coarse (right).

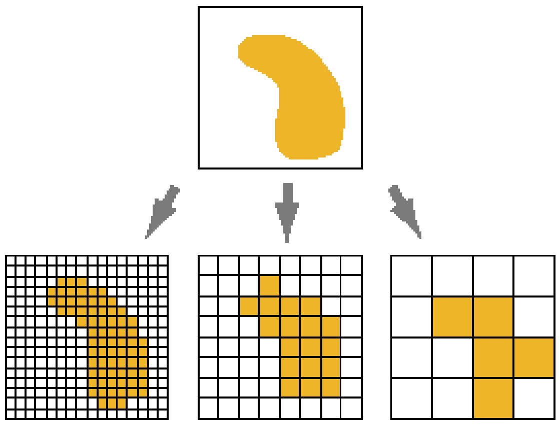

Figure 2.7.: Rasterization of a vector. Source: arcgis.com (2018)

Vector Data

Aside from storing data in a raster, it is possible to store it in mathematical vector functions. While raster data is very static, since the resolution has to be chosen when creating the file, vector data types are more flexible. Data is stored in a function that describes the location and the shape of data, which makes it ideal for concrete objects.
This permits storage of data that is needed. Coming back to the previous example, the location of a building would simply be stored by the specified coordinate. To better distinguish data they are grouped in features. A feature can have one of the following types:

Point:
Points are the most simple type of feature. It consists of one coordinate.Figure 2.8 presents the usage of points for wells.

LineString:
LineStrings consist of a number of points that are connected by a line that does not have to be closed. In the example image, it represents the river.

Polygon:
Polygons are like LineStrings, except that the shape has to be filled. The example images uses a polygon to display the lake.

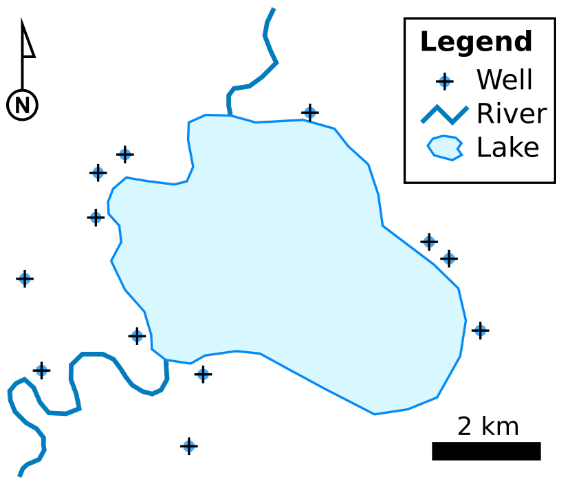

Figure 2.8.: Showcase of the different vector types. source: Toews (2007)

2.2. GIS Technologies

Given the properties of the mentioned projections and various coordinate systems with their different revisions in the previous section, it is necessary to use appropriate tools to reduce the complexity when Handling GIS. These include file formats, command line tools, libraries for programming languages and databases.

2.2.1. Formats

When working with GIS, it is necessary to have a reference to the used projection and coordinate system built inside the file. Over the years, many different types have been created, however, there are some very common types that are either fast, or convenient. The remaining part of the section introduces four formats, a text-based and a binary format for each vector and raster data.

GeoTIFF

Georeferenced Tagged Image File Format (GeoTIFF) is a binary raster file format, based on TIFF. It was created by N. Ritter and M. Ruth (1997) and the standard was defined by Niles Ritter et al. (2000). GeoTIFF is fully compatible to TIFF 6.0 and extends its tags to store geospatial information. It supports a vast variety of projections and coordinate systems. GeoTIFF supports different bands, like the ones explained in section 2.1.2.
The fact, that the format stores information in binary and relies on the TIFF compression

algorithms, results in a smaller file size, compared to formats like AsciiGrid. This advantage becomes particular valuable, when large areas of the image do not contain data or otherwise have high amounts of redundant values.
For the reasons mentioned, GeoTIFF is the most popular format for storing geospatial raster data.

Esri AsciiGrid

AsciiGrid is a text based raster file format created by the company Esri. In comparison to GeoTIFF it is a very simple format. The spatial reference system is EPSG:4326 and cannot be changed since it is not part of the file's metadata. Multiple bands and compression are also not supported.
The advantage of AsciiGrid is it's simplicity. The file can be created in a simple text editor and can be easily parsed. Listing 2.1 shows an example file. **ncols**, which specifies the number of values per row, while **nrows** define the number of rows. **xllcorner** and **yllcorner** specify the lower left corner of the image in degrees. The **cellsize** is also measured in degrees, it specifies the size of one pixel. the **nodata_value** defines a value that is considered to be of no information, it should be chosen outside the range of normal values. The listing below provides the data of the example. The delimiter between two data entries is a space.

```
ncols 4
nrows 2
xllcorner 10.001389
yllcorner 53.565278
cellsize 1
nodata_value -999
1 2 3 4
5 6 7 -999
```

Listing 2.1: An example AsciiGrid file.

Esri Shapefile

Shapefile is a binary file type for vector GIS data and was created by ESRI (1998). It supports the three data types **Point**, **LineString** and **Polygon**, which were mentioned in section 2.1.2. Besides storing the actual vector GIS, it is possible to store information in a data table for each feature. This allows to add information like size of a shape or population in an area. A Shapefile contains several files. **shp**, **shx** and **dbf** are mandatory and others are optional. The mandatory files and the most common optional file are listed below.

- The .shp contains the geometry with all the features.
- The .shx holds the index of geometries for better lookup performance.
- The dbf file is a sBASE database file that contains the data table.
- The projection is stored in the .prj file. While this is optional, it is usually supplied.

Today Shapefile is the most used file format for vector data. It is however, criticized for its limitations in length of names and number of entries in the data table, together with multiple files and a hard 2GB file size limit. A prominent replacement for the future could be GeoPackage (Consortium 2014).

GeoJSON

GeoJSON is a text-based format for storing vector GIS data. It is a convention based on JavaScript Object Notation (JSON) created by Howard Butler et al. (2008). It is also defined as the Internet Engineering Task Force (IETF) standard RFC 7946 (H. Butler et al. 2016). It supports all the features of a Shapefile without the limitations mentioned above. Due to it's text-based nature it is no replacement for a Shapefile since it has to be parsed. Another disadvantage is the missing compression, which causes file-sizes to be much greater.
Listing 2.2 shows an example of a GeoJSON file. In shows one layer (myLayer) with 3 features, one for each type.

```
{
  "type": "FeatureCollection",
  "name": "myLayer",
  "crs": {
    "type": "name",
    "properties": { "name": "urn:ogc:def:crs:OGC:1.3:CRS84" }
  },
  "features": [
    {
      "type": "Feature",
      "properties": { "id": 1, "name": "Point feature" },
      "geometry": {
        "type": "Point",
        "coordinates": [ 20, 20 ]
      }
    },
    {
      "type": "Feature",
      "properties": { "id": 1, "name": "LineString feature" },
      "geometry": {
        "type": "LineString",
        "coordinates": [[30, 10], [10, 30], [40, 40]]
      }
    },
    {
      "type": "Feature",
      "properties": { "id": 1, "name": "Polygon feature" },
      "geometry": {
        "type": "Polygon",
        "coordinates": [[[30, 10], [40, 40], [20, 40], [30, 10]]]
      }
    }
  ]
}
```

Listing 2.2: A GeoJSON example file.

2.2.2. Libraries

The following tools are most common, when working with GIS.

GDAL/OGR

The Geospatial Data Abstraction Library (GDAL) and OpenGIS Simple Features Reference Implementation (OGR) is the combination of two libraries with various comand-line tools that allows manipulation of geospatial data. GDAL manipulates raster and OGR vector data. In it's latest version (2.3.0) it support 154 raster and 93 vector formats, among them are the 4 file types mentioned before.
GDAL is an The Open Source Geospatial Foundation (OSGEO) project and the most broadly use command-line tool for manipulating GIS. Many libraries and applications rely on GDAL for their spatial implementations. Among them are Esri ArcMap, QGIS, GRASS GIS, Google Earth and the programming language R.
The library is capable of displaying detailed information about GIS files, which includes georeferencing, reprojecting and converting one type to another. It can also create tile maps and rasterize images.

NetTopologySuite

NetTopologySuite (NTS) is a ASP.NET implementation of the JTS Topology Suite (Java Topology Suite). It is a library for geospatial vector data manipulation and complies to the Simple Feature Access specification from Consortium et al. (2010) published by the Open Geospatial Consortium (OGC). NTS is one of the few options that support geospatial editing for ASP.NET Core.

GeoTools

GeoTools (Bledsoe, Brown, and Raff 2007) is a Java library for geospatial manipulation. It is an OSGEO project and complies to the OGC standards. The library supports various raster and vector formats for read and write operations. GeoTools is the most used GIS library for the Java programming language.

2.2.3. Storage Solutions

The following databases are most common in the GIS domain.

PostGIS

PostGIS is a spatio-/ temporal database, based on PostgreSQL. It supports storing vector and raster GIS data. Like PostgreSQL, the query language is SQL (Toups 2016).
The import process is done with specific tools that come as command-line applications which are bundled into the installation. The geometry type allows spatial queries regarding certain areas. These allow to extract single pixel values from raster files as well as feature extraction from vector files.
To read more about PostGIS and PostgreSQL see Obe and Hsu (2011) and Momjian (2001).

MongoDB

MongoDB is a document-based (NoSQL) database. Data is stored in Binary JSON (BSON). Input, as well as output data is in JSON format.
As of version 3.4.9 MongoDB supports vector GIS. It does not support a wide range of format types, but only GeoJSON (H. Butler et al. 2016). However, it is quite easy to convert vector files to GeoJSON using GDAL.

GeoServer

The GeoServer is a web-based application that allows storing and managing GIS data. It is not a database, but it can manage data storage in the file-system, as well as connect to PostGIS (Růžička 2016).
Using this software, it is possible to view the imported GIS files in a WebBrowser, using OpenLayers. It also allows searching, managing and converting data types. File conversion is done, using the OGC standard APIs Web Feature Service (WFS) and the Web Coverage Service (WCS).

2.3. Multi-Agent Research and Simulation (MARS)

MARS is the name of a working and research group at the HAW (University of applied Sciences) in Hamburg, Germany. It conducts in research on multi-agent simulations as described by Wooldridge (2009) and develops its own simulation platform.
The MARS system is build from several components and services that are bundled together in a micro-services infrastructure. Conceptually there are two groups that are to be distinguished, the LIFE simulation system and the supporting services around it. While LIFE is a monolithic application, the support services are split into micro-services and form, what is called the MARS Cloud System.

2.3.1. MARS Cloud System

The MARS Cloud is a collection of micro-services designed around the MARS LIFE simulation core. They handle all the preparative tasks, as well as analysis after the simulation.

Import Services

They handle the initial upload of all data into the system. This includes validating the input files, conversion between file types and finally the persistence of data.

Web UI

The UI is the center piece of the cloud system. This component provides the interface between the back-end services and the user.

Scenario Configuration

The Scenario configuration is responsible for combining the input data with the model, as well as defining the simulation parameters like the resolution, duration of a simulation and the number of agents.

Result Output

The „Result Output“ defines the data that is written to the database during the simulation. This data is later used for the visualization.

Data Analytics

During and after the simulation, this component provides real time data analysis and visualization of the data tagged for the result output.

2.3.2. MARS LIFE

MARS LIFE is a large scale, multi-agent simulation system developed by the MARS group (Hüning, Adebahr, et al. 2016; Hüning, Wilmans, et al. 2014). It is capable of handling up to several million agents (Hüning 2016) with interactions between each other. For a better scalability, it was designed with distribution in mind. Currently there are two ways of creating models for LIFE.

Models can be written in plain ASP.NET Core. This approach allows the model developer to use the full potential of the C# language and gives best performance and versatility.
Many domain experts are not proficient in writing code, so the use of C# is a barrier of entry. The MARS Domain Specific Language (DSL) created by Glake (2018) allows an alternative path that reduces the complexity for the user and allows the domain experts to create large scale simulations without them being expert programmers.

3. Analysis

The MARS system, as described in section 2.3 is a complex environment with many parts involved. This chapter describes the current state of the system and defines use-cases as well as requirements that the GIS layers have to fulfill. This approach is based on standardized GIS formats.

3.1. Current State

In order to use data inside a simulation, it has to be import via the WebUI. It is then processed by the appropriate service and made available for the simulation in LIFE. This thesis focuses on replacing the existing layer infrastructure with a generic, GIS-based approach. In order to fully understand the process, a brief understanding of the current system is needed.

Figure 3.1 shows the services and their relations. Data is uploaded through the WebUI into the File-svc. The File-svc is the central component for handling imports. It persists all files as GridFS inside the MongoDB and creates a metadata entry for the upload. Afterwards, it calls the responsible import service. That service requests the file, and handles it's type specific import. In the process the metadata is updated accordingly.
In case of a model upload, the reflection-svc stores the detected constructor types from the layers of the model in the Metadata Service.
In the next step, the user maps the uploaded data files with the model, according to the reflection results and defines global simulation parameters. This part is realized by the Scenario-svc.
The following step configures the result output, by using the Result Configuration Service.
Finally the Sim Runner Service starts the simulation.

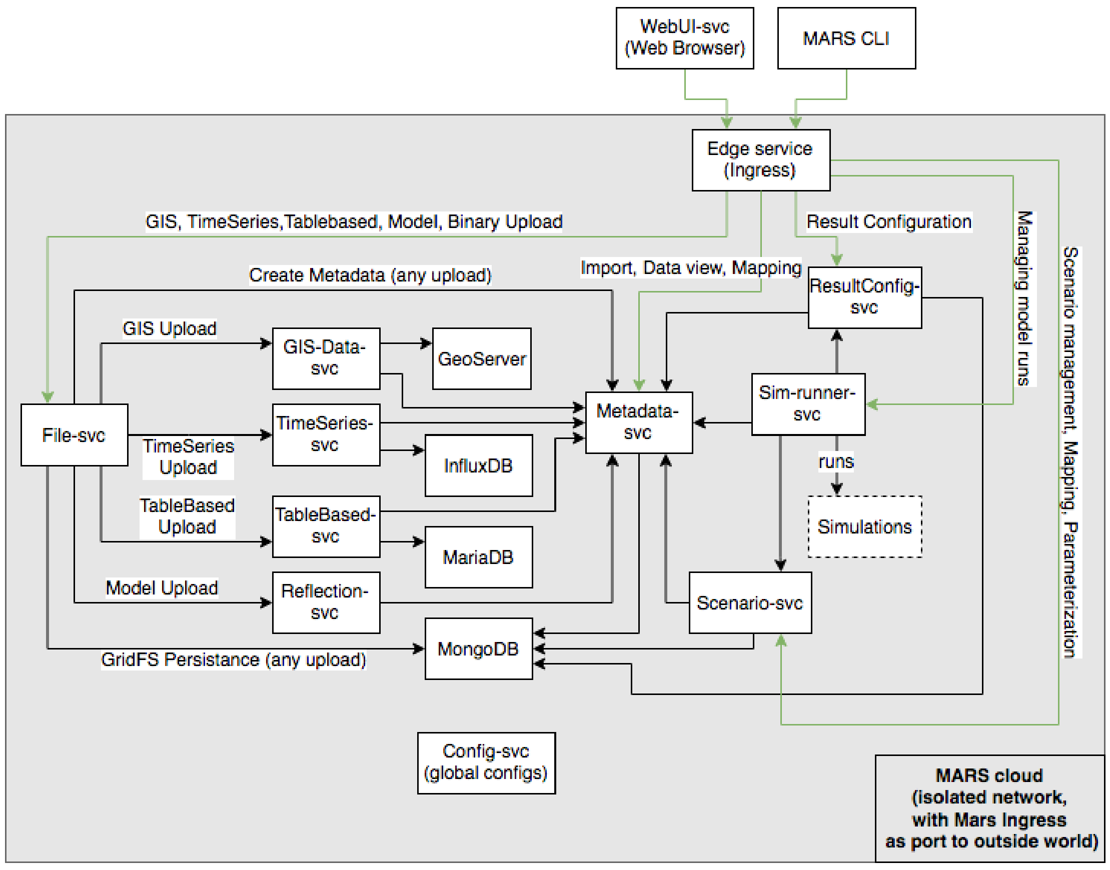

Figure 3.1.: Overview of the service infrastructure

The original idea was to have an easy process to add new layers to the system. Unfortunately the workflow described above, involves the developer to make changes to the following services and potentially create a new import service:

- Api-svc (Golang)
- File-svc (Java + Spring)
- Metadata-svc (Java + Spring)
- Reflection-svc (ASP.NET Core)
- Scenario-svc (Python)
- ResultConfig-svc (ASP.NET Core & TypeScript)
- WebUI-svc (TypeScript)

- A new Layer in LIFE (ASP.NET Core)

To sum up, the process of creating a new layer type for simulations involves the creation of one micro-service, including a new layer in LIFE, as well as altering 7 existing micro-services that were written by 6 people in 5 different programming languages.
The need for new layers was eliminated by creating a generic layer. The first version of this GIS-based layer was created by the author of this work (Karsten 2018).

3.2. Use-Cases

The existing layers in the LIFE system fulfill certain use-cases. The GIS layer is a generic layer, that does not only have to fulfill the existing use-cases, but also to be apply-able to new circumstances. The following list covers features of the old layers and includes new use-cases brought to attention by the model developers. These use-cases and the requirements which will be derived from them, are the bases for the implementation in chapter 5 and the experiences in chapter 6.

Use Case 1	**GIS type detection**
Scope:	File import
Level:	Raster and vector detection
Primary Actor:	User
Stakeholders and Interests:	• User: Wants to upload a vector file. • User: Wants to upload a raster file.
Preconditions:	• The user has a GIS file of one of the following types: GeoTIFF, AsciiGrid, Shapefile or GeoJSON • The file is geo-referenced.
Postconditions:	• The type of GIS is determined. • The process is transparent to the user.
Main Success Scenarios:	

1. The user uploads a file like a GeoTiff.
2. The type raster gets detected and is used in the simulation.

Use Case 2	**File type independence**
Scope:	File import
Level:	File conversion on import
Primary Actor:	User
Stakeholders and Interests:	• User: Wants to upload a vector file as a Shapefile or GeoJSON. • User: Wants to upload a raster file as a GeoTIFF or AsciiGrid.
Preconditions:	• The user has a GIS file of one of the mentioned types. • The file is geo-referenced.
Postconditions:	• The file gets converted to the right type. • The process is transparent to the user.

Main Success Scenarios:

1. The user uploads a file like a GeoTIFF.
2. The file gets converted to AsciiGrid.
3. It is used in the simulation without user interaction.

Use Case 3	**Obstacles**
Scope:	Simulation
Level:	Movement
Primary Actor:	Agent
Stakeholders and Interests:	Elephant: Wants to move within the Kruger National Park (KNP) without crossing the fence.
Preconditions:	• The elephant is a movable agent. • A obstacle layer exists. • Position is within bounds or outside.
Postconditions:	• The movement path does not cross the fence. • The elephant did not enter or leave the KNP.

Main Success Scenarios:

1. The agent wants to move to a position **inside** the fence.
2. The path does **not cross** the fence.
3. He **arrives** at the destination.

Alternative Success Scenarios:

1. The agent wants to move to a position **outside** the fence.
2. The path **crosses** the fence.
3. He cannot move.

Use Case 4	**Overcome Obstacles**
Scope:	Simulation
Level:	Movement
Primary Actor:	Agent
Stakeholders and Interests:	Elephant: Wants to go to KNP's fence if it is weak enough.
Preconditions:	• The elephant is a movable agent. • A obstacle layer exists. • His path crosses the fence. • The resistance value of the part of the fence is smaller than his strength.
Postconditions:	• The elephant crosses the fence. • His new position might still be inside the KNP.

Main Success Scenarios:

1. The agent wants to move to a position **outside** the fence.
2. The fence **cannot** hold him.
3. He **crosses** the fence.

Alternative Success Scenarios:

1. The agent wants to move to a position **outside** the fence.
2. The fence **holds** him.
3. He **does not cross** the fence.
4. He **cannot** move.

Use Case 5	**Sensing water**
Scope:	Simulation
Level:	Sensing
Primary Actor:	Agent
Stakeholders and Interests:	Elephant: Wants to sense the closest water source within a radius.
Preconditions:	• A water source layer exists. • The elephant is within range of a water source.
Postconditions:	The elephant knows were the next water source is.
Main Success Scenarios: 1. The elephant desires water. 2. He finds the closest water source within his range. 3. He walks towards that source.	

Use Case 6	**Sensing shadow**
Scope:	Simulation
Level:	Sensing
Primary Actor:	Agent
Stakeholders and Interests:	Elephant: Wants to find the closest shadow that covers him completely within his visual range.

Preconditions:	• A shade layer exists. • The elephant see the closest shadow with the desired coverage.
Postconditions:	The elephant knows were the next shadow is.

Main Success Scenarios:

1. The elephant seeks shadow.
2. He finds a shadow with enough coverage.
3. He walks towards that source.

Use Case 7	**Consume Biomass**
Scope:	Simulation
Level:	Parallel write access
Primary Actor:	Agent
Stakeholders and Interests:	Elephant: Wants to consume biomass in his area.
Preconditions:	• A biomass layer exists. • There is biomass left.
Postconditions:	• The biomass is reduced. • The value does not go below zero.

Main Success Scenarios:

1. The elephant reduces the biomass at his position.
2. Other agents do the same at the same time.
3. No biomass is consumed twice.

Alternative Success Scenarios:

1. The elephant tries to reduce the biomass at his position.
2. No biomass is left.
3. Nothing was consumed.

Use Case 8	**Time series**
Scope:	Simulation
Level:	variable values over time
Primary Actor:	Agent
Stakeholders and Interests:	Tree: Wants to grow based on precipitation.
Preconditions:	• A precipitation layer exists. • There is data for the current time.
Postconditions:	The tree grows.

Main Success Scenarios:

1. The tree requests the precipitation for the current time.
2. It grows accordingly.

Use Case 9	**Distance calculation**
Scope:	Simulation
Level:	Abstract from projection
Primary Actor:	Agent
Stakeholders and Interests:	Elephant: Wants to move towards a GPS coordinate.
Preconditions:	The target gps coordinate exists.
Postconditions:	The distance in meters is known.

Main Success Scenarios:

1. The Elephant knows about a water source and wants to determine it's distance.
2. The layer calculates the distance and converts it from degree to kilometers.
3. The Elephant starts walking.

3.3. Requirements

The use-cases in the previous section focus mainly on the model developers desired features. Apart from those, there are requirement determined by the environment, the final application has to run in. The following section combines all requirements into functional and non-functional.

3.3.1. Functional Requirements

F1 GIS Import: **Has to** be able detect the CRS.

F2 GIS Import: **Has to** automatically converted raster GeoTIFF into AsciiGrid.

F3 GIS Import: **Has to** automatically converted vector Shapefiles into GeoJSON.

F4 GIS Import: **Has to** be integrated into the MARS Cloud.

F5 GIS Import: **Has to** be able to handle any type of input as a Zip file.

F6 GIS Import: **Should** be able to detect the GIS type (raster or vector) transparent to the user.

F6.1 This reduces the necessary to specify the type on upload.

F7 GIS Import: **Should** interpret files as WGS 84 if no CRS could be detected.

F8 GIS Layer: **Has to** be able to calculate if a vector feature is contained by another.

F9 GIS Layer: **Has to** support obstacles that allow to limit agent movement.

F9.1 Different types of obstacles have to be supported.

F9.2 Variant strengths of obstacles are needed.

F10 GIS Layer: **Has to** to be able to calculate the closest coordinate that meets a criteria, around a specific point.

F10.1 Find the highest adjacent coordinate (needed for potential-field maps).

F11 GIS Layer: The altering of provided data **has to** be supported.

F11.1 Parallel access needs to be possible and scale with increasing amounts of agents.

F12 GIS Layer: Time-series data **has to** to be handled by the layer

F13 GIS Layer: Distance calculation between points **has to** be supported.

F13.1 Return distance in degree.

F13.2 Support conversion to kilometers.

F14 GIS Layer: **Has to** be able to read data at a position.

F14.1 GPS.

F14.2 Cartesian.

F15 GIS Layer: **Has to** be able to calculate if vector features intersect.

F15.1 Complex paths need to be supported.

F16 GIS Layer: **Has to** be able to calculate if vector features are inside other features.

F17 GIS Layer: **Has to** be integrated into MARS LIFE.

3.3.2. Non-Functional Requirements

NF1 GIS Layer: **Has to** be written in ASP.NET Core.

NF2 GIS Layer: **Has to** not diminish the read performance by more than 10% in any scenario.

NF3 GIS Layer & Import: **Have to** be tested with unit tests, to allow extensibility for future changes.

NF4 GIS Layer: **Should** Errors are handled to prevent simulation crashes.

NF5 GIS Layer: **Should** Log messages are written to standard out.

4. Software Design

This chapter covers the process of finding the right design and technologies to fulfill the requirements gathered in the previous chapter. The Solution Statements cover possible ways of solving the problem. The last section covers the final design.

4.1. Methodology

Based on the requirements a technical research has been carried out to find the right technologies. These findings have been compared by features and then tested in terms of performance. The technologies required for the import and the GIS layer implementation during the simulation, have to fulfill different requirements (section 3.3). This is why they are looked at independently in the following sections

4.2. Solution Statements – Import

The File Import needs to detect the type of GIS that was uploaded, handle zipped data, determine the coordinate system, fall back to WGS 84 if none was detected and convert data to the right type. This final result needs to be integrated into the MARS Cloud.

4.2.1. Detection of the file type

Detection of the correct GIS type can be done by checking the file ending. Files in a zipped container have to be extracted before doing so. The process is very basic and therefore can be done in any technology. The file formats with their corresponding endings are listed below:

Zip files:

- .zip

GeoTIFF:

- .tif

- .geotiff

Esri AsciiGrid:

- .asc

GeoJSON:

- .json
- .geojson

Shapefile:

- .shp
- .shx
- .dbf

13 additional, non-mandatory file suffixes exist. For more details, refer to ESRI (1998).

4.2.2. Validating File Types & detecting CRS

Once the type of file has been determined, it has to be validated and the coordinate system needs to be detected. This requires the program to be able to read and understand the detected type.
As described in section 2.1, a large amount of projections and coordinate systems are possible. Implementing those types would be an act of reinventing the wheel, so an existing library will be used to handle this task.
There are libraries for a few languages, but since GIS is a very specialized field, many of them lack essential features. The biggest set of proper tools are offered for „R“ and „Python“. For Java there is one feasible option, called GeoTools (see section 2.2.2).

4.2.3. Converting File Types

Due to the shortcomings of GIS libraries, there is currently no way to read GeoTIFF inside MARS LIFE, which is written in ASP.NET Core. Therefore the files have to be converted to a supported format. This conversion can be done in a variety of ways. The methods that were taken into closer consideration are the GeoServer and GDAL. In the following paragraphs, the two are being compared by features and performance.

GeoServer

The GeoServer, as introduced in section 2.2.3 that allows the conversion of types. Unfortunately it is done via the WFS API which does the conversion upon retrieving the file. While this makes the process of retrieving files very flexible, it leads to recurring conversions for multiple simulations with the same file.

GDAL/OGR

GDAL, as described in section 2.2.2 is a C library with bindings for all common languages. Compared to the GeoServer, it is a more light-weighted solution, since it does not introduce it's own service.

Persistence

Currently the GeoServer is being used for GIS persistence. It has many more features, but the only ones being used are the persistence and file conversion capabilities. For more detail check the previous work of the author (Karsten 2018).
The workflow of the current import already persists data before it is handed over to the GIS import service. This allows to make the GeoServer obsolete by replacing the stored file with the converted version.

Performance

The main reason the GeoServers usage was questioned in the past, is it's poor performance in certain scenarios (for further information refer to Karsten (2018)). This is why a benchmark was created that measures the execution time of the technologies GIS file conversion.
The performance test was done with 3 vector Shapefiles and 3 raster GeoTIFF files of small, medium and large size. The Shapefile was converted to GeoJSON and the GeoTIFF to AsciiGrid. The tests were done three times. Figure 4.1 and 4.2 show the averaged results in seconds.

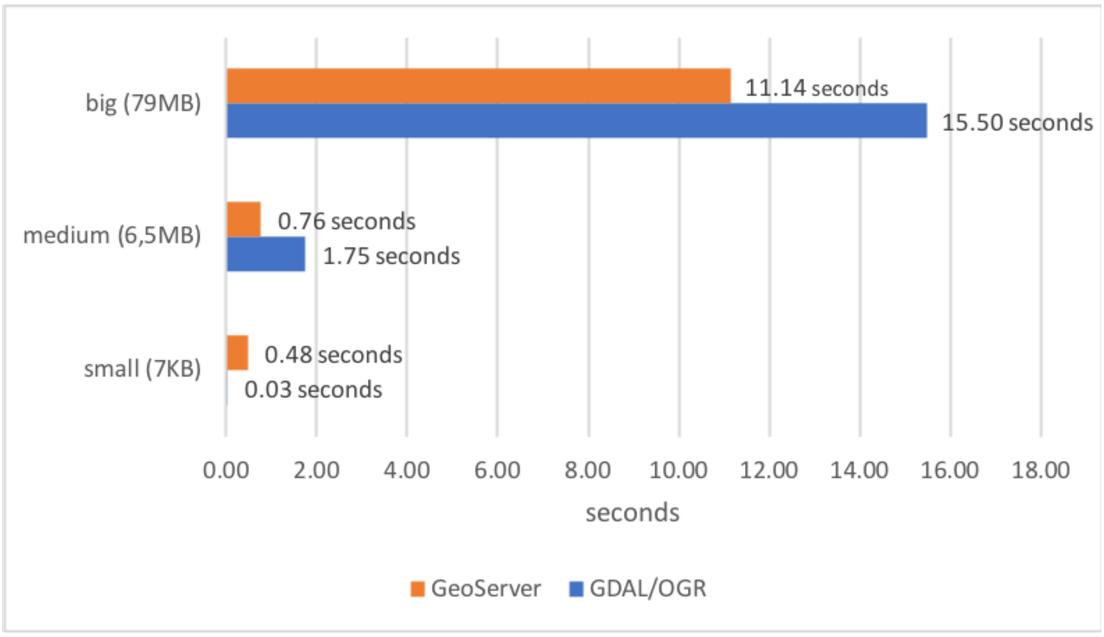

Figure 4.1.: GeoServer vs. GDAL/OGR Vector conversion

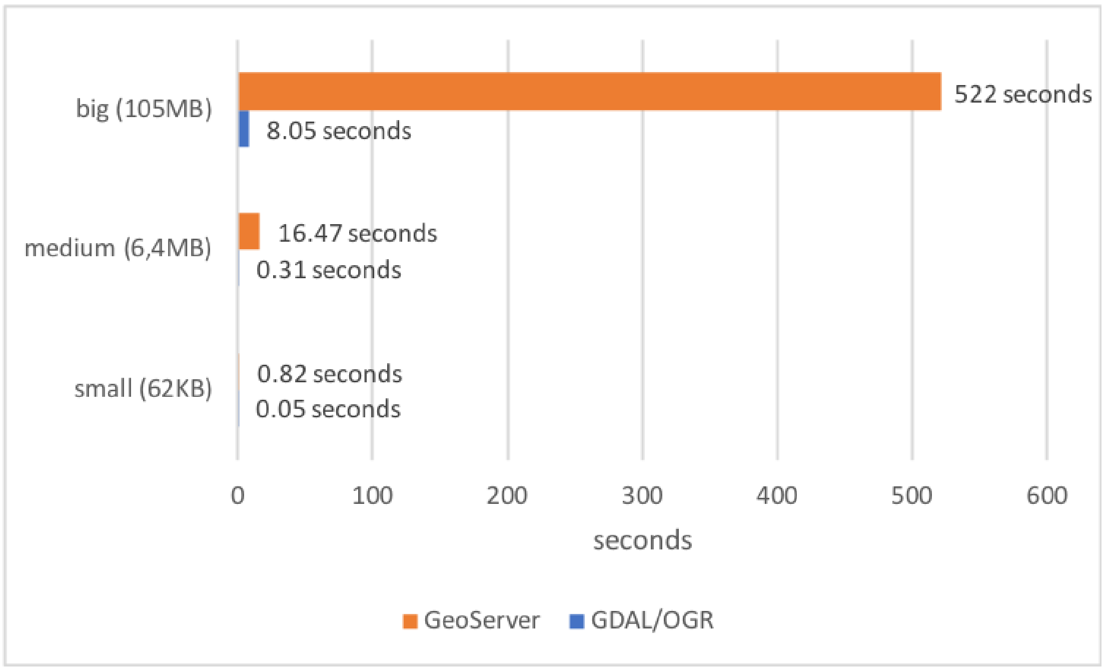

Figure 4.2.: GeoServer vs. GDAL/OGR Raster conversion

Figure 4.1 shows the vector conversion results. Surprisingly the performance of the GeoServer was better than GDAL, when converting the shapefiles. This is especially worth noting, since the data had to be send over the network from a remote server. The reason for this is that the GS runs as a separate service. The network latency also explains why the GeoServer is slower, when converting the small file. Overall the performance of the GS is better, but not by a huge margin.

The conversion of raster data in figure 4.2 shows entirely different results. GDAL is about 16 times faster on a small file, 53 times faster on the medium file size test and almost 65 times faster on the large file. The conversion of the big file took over 8.5 minutes compared to 8 seconds with GDAL. The time complexity of the GeoServer's algorithm is $O(2N)$. As a result, converting a file of 1GB in size would take an estimate of 2.76 hours, which is not acceptable.

4.3. Solution Statements – GIS Layer

The GIS layer is a data layer inside the MARS LIFE simulation system, as explained in section 2.3.2. This means that the GIS component has to be written in ASP.NET Core.
The raster and vector component are able cover a similar functionality, but they approach a given problem in different ways. Raster GIS is less complex and therefore generally faster, but also less accurate. Vector GIS is more precise but relies on complex algorithms to function.
The data structure and the libraries used to read and process the raster and vector data vary by big margin. For this reason the two components have been separated into individual components.

4.3.1. GIS Raster Layer

The most fundamental functionality of the layer is the capability to read values from geo-referenced raster files. The constraints are explained in section 2.1.2.
This requires the functionality to interpret the given file. The implementation of a predicate based cell selection algorithm is also required, as well as the support for time-series data.

File parsing

GeoTIFF would be the preferred format, as it stores data in binary. However, there is currently no library that supports reading the format. Creating a parser is out of scope for this thesis, due to the following reasons.
Implementing a parser requires decoding the TIFF compression algorithm and adding the

spatial logic on top. Neither exist in the standard or any other GIS library for ASP.NET Core, since it is no trivial task. The performance gain on the other hand would only effect the time of initialization which is done once for every simulation.

The alternative is using an AsciiGrid file, as described in section 2.2.1. Currently, there exists no library for reading AsciiGrid in ASP.NET Core however, the format is text-based, has a simple header, does not support multiple bands or coordinate systems and no compression algorithms. This results in a reduced complexity to write the parser, which makes it worth the afford.

Potential Field

Potential field data are pre-generated files that allow spatial routing to a certain cell based on the highest adjacent cells. This is needed for detecting the shortest path on raster files. The expensive part is the creation of the layer, which is done ahead of time and the lookup times are fast.
The current MARS use-case are agents, sensing water in the distance. During runtime, it requires to check the adjacent cells in order to get the highest value. This is done repeatedly until the destination, e.g. 100% is found. The number of checks for every step is $n <= 8$.
Figure 4.3 shows a potential field image for water sources. The bottom right cell has a value of 100% black, indicating a water source. The gray areas are the area an agent can sense water from. The white area is outside the sensing reach of the agent.
To navigate to the water source, the layer has to provide the capability to find the closest adjacent cell. The agent can then go there, if it wishes to drink.

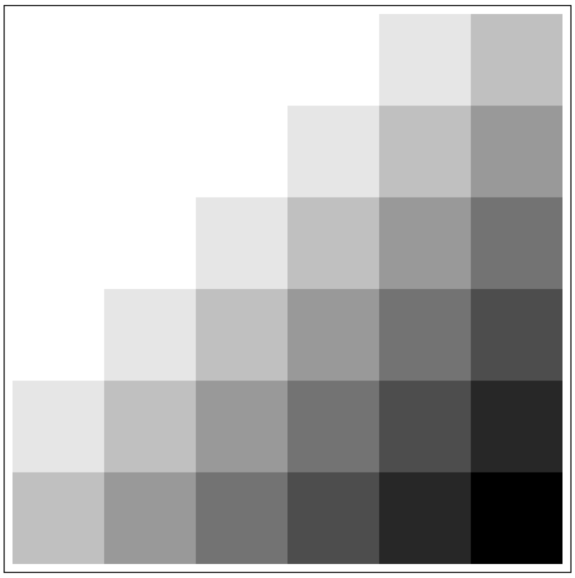
Figure 4.3.: Example potential field

Finding the closest Cell

In the first version of the calculation, as specified by the author of this work, it was only possible to find a cell with specific values. This has been changed to predicate based logic, allowing the user to pass a boolean function into the layer. This allows to not only check for a specific value, but also to verify or check value ranges and more complex operations.
The function is evaluated and returns true, if the correct cell is found. This is done from the origin cell outwards until the maximum distance is reached or no cells are left to evaluate. In the example shown in figure 4.4 the origin cell contains a 0. It is located in the third row of the first column. The target cell is marked black. Each iteration, called level of the algorithm is labeled with a number in the image.
For each level, a list of surrounding cells is generated and the predicate expression is evaluated against those cells. If no match is found, a new list with cells, surrounding the previous one is generated and reevaluated. If a match is found the coordinates of the cell are instantly returned. On the forth level, the black target cell matches the expression. The coordinates are returned as a result, which terminates the algorithm. The last cell on level 4 is not evaluated.

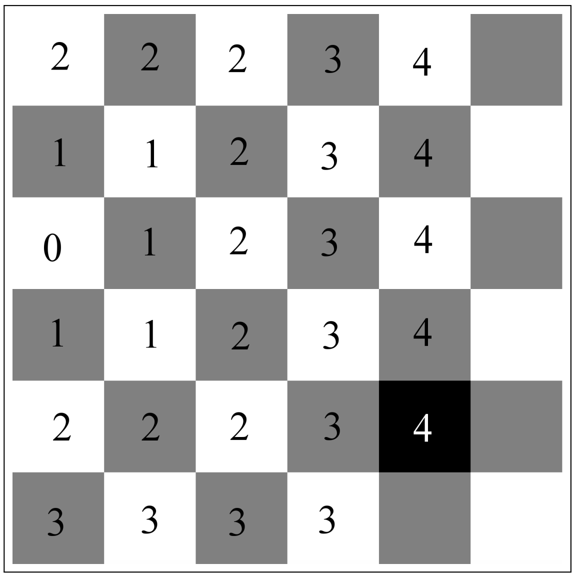

Figure 4.4.: Find a Cell by pattern matching

Time-series

Time-series data changes over time. There is a need to have changing layers over the duration of the simulation. AsciiGrid does not support more than one band like GeoTiff (see section 2.2.1), therefore multiple files have to be used and swapped during runtime.
The LIFE systems design strictly separates model code and initialization data. This means that a given model can be executed with different input files. The implementation of the logic to swap out files in the layer code, would break with this paradigm. Therefore the layer has to swap files based on the input.
The design of the MARS system allows to only pass and map one file to a given layer, so the raster files have to be combined in a zip-file. To control the duration of the validity, time-series raster files have to contain a „metadata.csv". This file has two columns. A time stamp that marks the beginning of validity and the corresponding file name. The example in listing 4.1 shows biomass data for the years 1979 and onwards. Each year has its own file. For the year 2000 the first file is being reused.

```
Date;File
1979-01-01T00:00:00;1979.asc
1980-01-01T00:00:00;1980.asc
1981-01-01T00:00:00;1981.asc
```

```
...
2000-01-01T00:00:00;1979.asc
...
```

Listing 4.1: An Example metadata.csv.

Parallel write Access

Parallel write access on files requires logic that takes care of conflicts, alternatively each file has to be locked during a write access. This is not an option for performance critical operation with >1 million parallel writes. Instead, a value based locking mechanism has been introduced. The values are stored in memory inside a thread safe data structure that handles each field individually. To guarantee the isolation of changes, it is important that the structure is not exposed directly to the user. Parallel threads that read the value and set it to the new value could create inconsistencies that are hard to debug.

4.3.2. GIS Vector Layer

The vector layer allows geo-spatial requests on desired files. As mentioned before, it covers basically the same features as the raster layer above. Additionally it offers geo-spatial operation regarding shapes and their interactions.

File Parsing

The file parsing is done with NetTopologySuite. This is a result of the performance tests done by the author of this work ahead of time (Karsten 2018). The default for vector data is the Esri Shapefile. However the compatibility in NTS was removed when the library was migrated to ASP.NET Core and this feature is still not available yet. The parsing of GeoJSON is supported and works without any restrictions.

GIS Operations

As defined by the requirements in section 3.3.1 certain GIS features have to be supported. The API has to expose methods that allow these calculation. Among others, these involve checking if paths intersect and rather a feature is inside another. Distance calculation between points is also required.

Nearest Neighbor Calculations

The default nearest neighbor calculation of NTS iterates over every point in every feature and then calculates the distance between all of them. This calculation takes along time and there is no way to terminate the process, once the closest point has been found. This means the algorithm always compares every feature with every other, making the algorithm have a $O(N^2)$ complexity class in any case.

An alternative provides the k-dimensional (k-d) tree algorithm by Bentley (1975). It offers excellent performance for nearest neighbor lookup on static positioned points. The GIS vector layers cannot change during the simulation, which makes it a perfect fit.
The algorithm is a space-partitioning data structure for organizing points in a k-dimensional space. The implementation is based on a multidimensional Binary search tree (BST). Inserts and lookups have an average complexity of $O(\log n)$ and $O(n)$ in the worst case.

Time-series

It is possible to store time-series data inside GeoJSON and the Shapefile's data table. Unfortunately the dBase that is used for the Shapefile data table has very small limits in terms of characters and number of columns that can be added. Removing compatibility with Shapefile is also not an option.
An alternative is the use of a simple csv-file that is stored inside the files zip-container and gets interpreted during initialization. This file is similar to the metadata file for raster time-series data, but here only one file is needed, since there was no use-case for multiple points on a certain timestamp. Listing 4.2 shows an example.

```
date;precipitationInMm
1989-01-01T00:00:00Z;14.3
1989-02-01T00:00:00Z;286.3
1989-03-01T00:00:00Z;57.6
1989-04-01T00:00:00Z;6.8
```

Listing 4.2: An example vector time-series csv-file.

Parallel write Access

The write access regard only the data table, since there are currently no plans to change vector files during runtime. Therefore the data table must be stored in a concurrent data structure for fast memory reads and writes that provides thread safe access.

4.4. Final Design

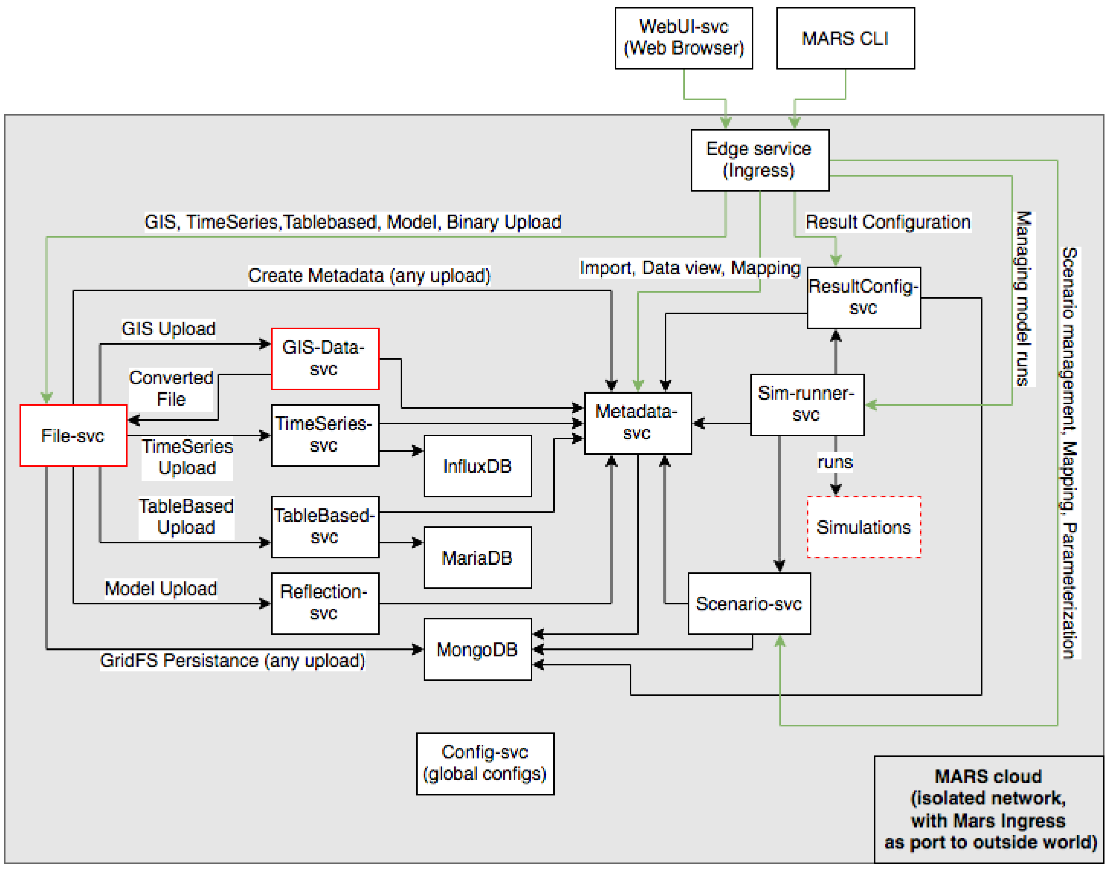

Figure 4.5.: New MARS Cloud Service overview

The previous sections discussed the design decisions based on the requirements in chapter 3. This section sums up the previous detailed explanations into the final design. Figure 4.5 shows the new service overview. The services surrounded with a red box have been created or changed by this design.

4.4.1. GIS Data Service

The GIS Data Service handles the import of any GIS based imports. It covers the following tasks:

1. Extract files from zip container.
2. Detect the type of GIS.

3. Check the validity of the files.
4. Convert raster files to Esri AsciiGrid and vector files to GeoJSON.
5. Delete not supported files.
6. Compress the converted files.
7. Persist the converted files.
8. Update the files metadata entries.

Steps 4 to 7 are optional and only happen, if the file has to be converted.
The service is written in Java, because GeoTools is a well supported solution for working with GIS data. The application will use the Spring Boot framework to create a web application. It is required to expose a Representational State Transfer (REST) API for establishing communication with other services.

Persistence

The persistence was previously handled by the GeoServer, but due to problems discussed in section 4.2.3 and 4.2.3 it has been removed. Files are persisted by the File Service upon upload. It's REST API has been adjusted, so the converted files could be stored in the Service's persistence. This removes one service from deployment and reduces friction between different import types.

4.4.2. GIS Raster Layer

The raster layer is a component inside the LIFE simulation system that is initiated with a metadata ID of a AsciiGrid raster file. Upon initialization it retrieves the actual file or multiple files in case of time-series data.
The file is parsed using the AsciiGrid Parser described below. The Layer exposes a ASP.NET Core API that fulfills the requirements defined before. All endpoints are documented in order to be used by model developers. A full API description can be found in section 5.2. No further dependencies are required for the layer to work. The handling of time-series data is done automatically, so the user will always get the proper data for the current simulation time.

AsciiGrid Parser

Due to missing core libraries for image processing, there currently is no library that support any common GIS raster format. This is why a parser for Esri AsciiGrid has been created.

It copies the data from each cell of the file into a concurrent data structure to allow parallel access on the file. This approach allows to reduce the scope of potential collisions from the file down to a single value, which decreases the amount of locking and increases performance.

4.4.3. GIS Vector Layer

The GIS vector layer is initialized just like the raster layer, except that it uses a GeoJSON file to initiate. The layer also stores its data in-memory, but since the vectors cannot change, thread safety is not a concern. The vector layer's data table on the other hand can change, therefore it is held in a parallel data structure like the raster data.
The Layer uses NTS as a data abstraction library to be able to support a wide range of geo-spatial coordinate systems and GIS algorithms and calculations.

5. Implementation

This chapter covers the implementation of the previously defined design. Each of the GIS Data Service components, raster and vector layer are looked at individually. For each service the implementation details are discussed, a sequence diagram is shown and the detailed API reference is shown.

5.1. GIS Data Service

As mentioned before, the GIS Data Service is written in Java. It uses Spring Boot to configure the Spring framework, which is used for the REST API. GeoTools is used for the interpretation of GIS files and the file conversion is done by GDAL. The used versions are listed in the following table.

Technology	Version	Release Date
Java	10	2018-03-20
Spring Boot	2.0.2	2018-05-09
GeoTools	19.1	2018-05-21
GDAL	2.3.0	2018-05-04

5.1.1. Build Process

The application uses maven as a configuration and dependency management tool and the source code is checked into the the department's GitLab. The process for building the application locally does not require the user to follow any specific steps however, there are a few considerations, when developing and deploying the service.

Build and Run Locally

It is possible to start the service locally. To do so, the JAR can be compiled using the following command:

```
mvn clean package
```

Listing 5.1: Build the application

This downloads all the dependencies, compiles the application into a Java Archive (JAR) and runs the unit tests. Afterwards the app can be started using:

```
java -jar target/gis-data-svc-2.2.*.jar
```

Listing 5.2: Run the application

This allows to start the service, run the applications unit tests and do manual tests of it's REST API. When testing against other services, it is useful to deploy the service in the beta deployment of the MARS group.

Build and Run inside the MARS Beta Deployment

The way this can be done is by changing the deployed version in the beta to dev, push a Docker image to the GitLab registry and apply the changes.
First the „gis-data-svc.yml“ file inside the „mars-beta“ has to be altered.

```
- image: <registry>/mars/mars-gis-data-svc/gis-data-svc:28143
+ image: <registry>/mars/mars-gis-data-svc/gis-data-svc:dev
```

Listing 5.3: Change gis-data-svc version

This change has to be applied using

```
kubectl -n mars-mars-beta -f gis-data-svc.yml
```

Listing 5.4: Apply changes to Kubernetes

Afterwards the application has to be build, the images needs to be pushed to GitLab and the service in the cluster has to be restarted. To streamline the process the following script can be used. It can also be found in the „start.sh“ script inside the repository.

```
DOCKER_REGISTRY="docker-hub.informatik.haw-hamburg.de"
PROJECT="mars/mars-gis-data-svc"
SERVICE_NAME="gis-data-svc"

mvn clean package
```

```

docker build -t ${DOCKER_REGISTRY}/${PROJECT}/${SERVICE_NAME}:dev .
docker push ${DOCKER_REGISTRY}/${PROJECT}/${SERVICE_NAME}:dev

kubectl -n mars-mars-beta delete pod -l service=${SERVICE_NAME}
```

Listing 5.5: Apply changes to Kubernetes

5.1.2. API Reference

This section discusses the REST API of the GIS Data Service and the changes made to the File Service.

File Service: POST /files/replace

The File Service has one new endpoint for replacing converted files, which is shown by figure 5.1. It's purpose is to allow to replace files during the import. This is used by the GIS Data Service, when it converted a file.
The other endpoints are not shown here, since they are not part of the authors work.

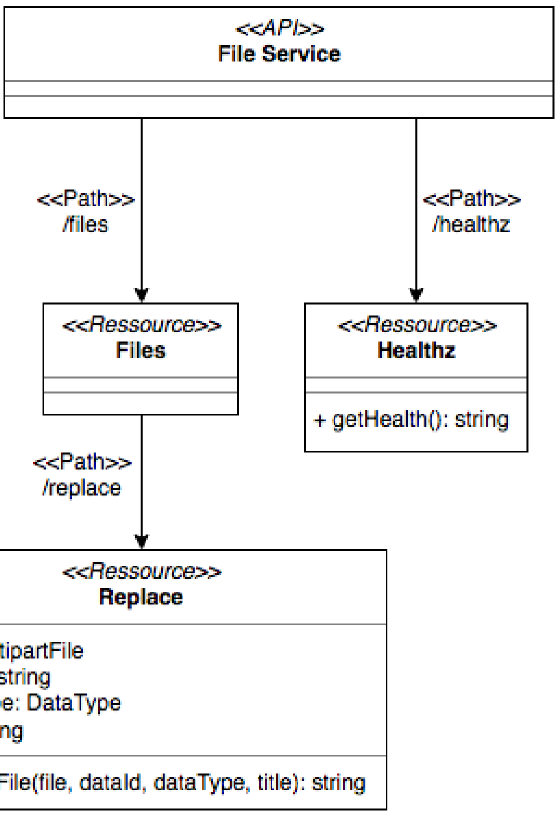

Figure 5.1.: The File Service's REST API

GIS Data Service: POST /gis

The changes that removed the GeoServer also eliminated the necessary for retrieving data from the GIS Data Service. This is because, persistence is handled by the File Service. Therefore the GIS Data Service has only one endpoint left. This endpoint is being used by the File Service to trigger the import. Figure 5.2 shows this endpoint. It requires to specify the dataId, the name of the file and its title.

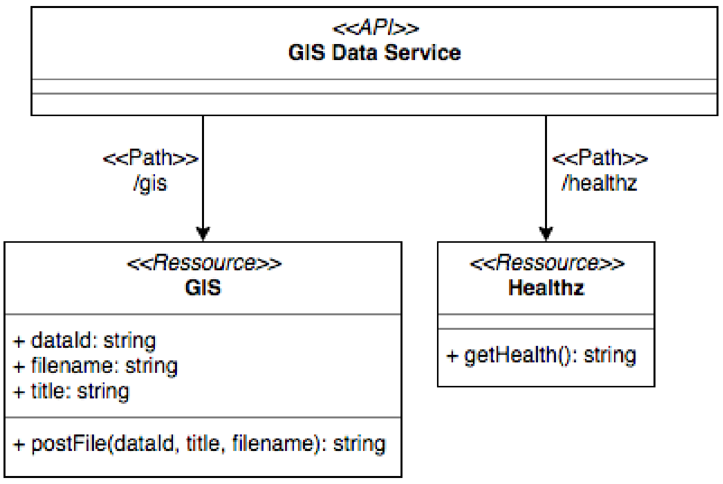

Figure 5.2.: The GIS Data Service's REST API

Status Codes:

Both services have the same status codes for handling responses. They are listed in the following table.

Code	Description
200	OK.
201	Created
400	Wrong parameter
404	Not Found

5.1.3. Sequence Diagram

Figure 5.3 covers the whole process of importing a GIS file. The user in the WebUI triggers an import. This uploads the users file along with the form data to the File Service. It first inserts the file into MongoDB and creates the metadata entry. Afterwards the control is handed to one of the import services, in this case the GIS Data Service.
The service does its detection of the GIS type and conversion of files, like described before. In case a file has been converted, it will be send to the File Service, to replace the originally uploaded file. In order for this functionality to work, a new endpoint, called „/files/replace", was added to the File Service's REST API.

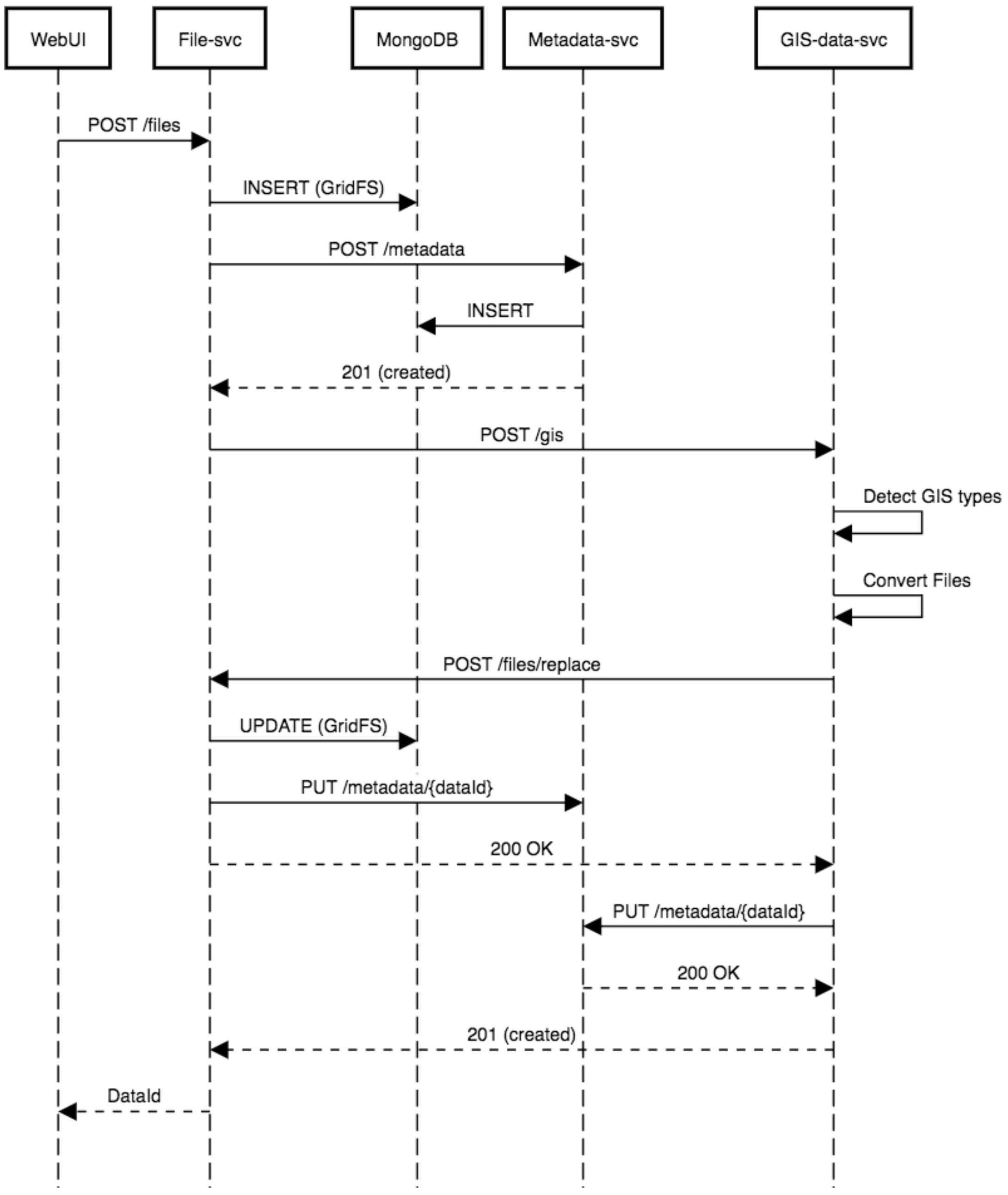

Figure 5.3.: Sequence diagram of the GIS import process

5.2. GIS Raster Layer

This section documents the implementation details of the GIS Raster Layer. It covers the use of time-series data, the AsciiGrid parser and a documentation of the API.

5.2.1. Time-series Data

During initialization, the time-series metadata file is parsed and sorted by date. The result is a list of dates with their corresponding time-series files. This means that whenever the time advances to a point, where the file has to be changed, the index reference has to be incremented by one.

In every tick of the simulation, the current timestamp (n) is calculated. The layer then simply has to compare this timestamp to the next ($n + 1$) in the list. If the next timestamp is larger, meaning later in time, nothing has to be done. In case the next timestamp is before the current file, it checks to make sure the $n + 2$ timestamp is after the current time to prevent errors with time-series files that do not line up with the tick duration.

If the new time-series file differs from the old, it is automatically loaded and parsed without the model developers influence. This removes a potential error source, but also does not give the user the possibility to influence file loading.

5.2.2. AsciiGrid Parser

The parser is part of the layer and serves the purpose of reading the AsciiGrid file. It also transfers the data to a concurrent dictionary for parallel write access. The data is stored with an IGridCoordinate as a key. This allows to directly access the field at any time. Methods for converting GPS coordinates to grid coordinates and vice versa are supported to improve usability.

5.2.3. API Reference

The following section explains the most important API endpoints, as they can be found in „LIFE-API / Layer / GIS / IGISRasterLayer.cs“ inside the LIFE repository. A full list of endpoints and the metadata API can be found in appendix section A.3.1.

```
IReadOnlyDictionary<IGridCoordinate, double?> Grid { get; }
```

Listing 5.6: Raster API – IReadOnlyDictionary

The Read only dictionary allows direct access to the grid representation. It is meant for result output or debugging during development.

```
IAscMetadata Metadata { get; }
```

Listing 5.7: Raster API – Metadata

The grids metadata can be accessed the same way the actual grid can be.

```
void AddToGridField(IGridCoordinate coord, double value);
void AddToGridField(IGeoCoordinate gps, double value);
```

Listing 5.8: Raster API – AddToGridField

This method allows the user to increment or decrement values in a way that is thread safe. The grid cell can be provided via a Cartesian grid coordinate, specified in x and y or using a GeoCoordinate, using latitude and longitude. In case the specified cell is outside the bounds, a „ArgumentOutOfRangeException“ is thrown, as this is usually a user error that should not be suppressed.

```
double GetValue(IGridCoordinate coord);
double GetValue(IGeoCoordinate gps);
```

Listing 5.9: Raster API – GetValue

„GetValue“ is the central method which allows retrieving a value at a specific location. Just like the previous method, it throws an „ArgumentOutOfRangeException“ if the cell does not exist in the grid.

```
IGridCoordinate GetNeighbourCellWithMaxValue(IGridCoordinate positon);
IGeoCoordinate GetNeighbourCellWithMaxValue(IGeoCoordinate positon);
```

Listing 5.10: Raster API – GetNeighbourCellWithMaxValue

This method retrieves the adjacent cell with the highest numeric value. This allows the potential field files to be used.

```
IGridCoordinate GetClosestCellWithValue(IGridCoordinate coord,
Func<double, bool> predicat, int maxDistance);
IGeoCoordinate GetClosestCellWithValue(IGeoCoordinate gps,
Func<double, bool> predicat, int maxDistance);
```

Listing 5.11: Raster API – GetClosestCellWithValue

This method allows the detection of the closest cell with a value, matching a predicate. This is used for exploration. Due to the capability to pass a function as a parameter, it is possible to define e.g. ranges of values.

5.3. GIS Vector Layer

This section describes the implementation of the GIS Raster Layer. It features the use of NTS, the solution to the nearest neighbor lookup times, the handling of time-series data and a description of the major API endpoints.

5.3.1. NetTopologySuite

The NetTopologySuite was used to implement the GIS functionality. As shown in the following table, the currently used version is still a pre-release. This is due to the fact that current stable versions are still not compatible to ASP.NET Core. However the version did not cause issues during development, testing or finally usage.

Technology	Version	Release Date
NetTopologySuite	1.15.0-pre063	2018-05

5.3.2. Initialization Process

During initialization the whole GeoJSON file is read and parsed to a collection of features. This is held in memory and is being used throughout the simulation.
The Data table gets extracted from the feature collection into a concurrent dictionary to allow the desired parallel write access. In case, time-series data exist, it gets parsed afterwards, for details see section 5.3.3. Finally, a k-dimensional tree is created for increased nearest neighbor lookup performance (see section 5.3.3).

5.3.3. Time-series Data

The process of detecting the time-series value for the current tick works exactly like explained in the raster layer section. However, while the time-series raster data requires a reinitialization of the whole layer, this is not required in case of the vector layer. This is because the data that changes is a single double value for each timestamp. This means that instead of storing the file that will be used at a given point in time, the actual value is being stored and returned.

5.3.4. Nearest Neighbor Detection

As mentioned before, the default nearest neighbor detection of NTS is very inefficient. The reason for this is that several features can exist, which can contain of multiple points. This information is stored in nested lists which have no order. Therefore the distance calculation

has to compare the current point with every existing point in all features. The complexity in any case is $O(n)$ where „n“ is the number of points.
The algorithm stores the currently closest point and it's distance. On every iteration the new result is compared to the closest one, until all coordinates have been checked. Then the current closest point is returned, if it is within the max distance.
In a unit-test, using a river GeoJSON from the KNP model with 7,985 features, resulting in 24,593 single points, the execution time was 70 ms per calculation. The mentioned model has 15,000 elephants executing this lookups every time they searched for water. This resulted in greatly increased tick durations, compared to the previous implementation, using a non-GIS layer.

As a result of research, the k-d tree was chosen. The reason for this, is the excellent nearest neighbor detection for static points. During initialization of the simulation, all points of every feature have to be added to the tree. In comparison to the previous implementation, this had to be done only once.
During the simulation the nearest neighbor lookup is done by traversing the tree. This allows nearest neighbor detection without having to search every node, resulting in an average complexity of $O(\log n)$. The distance calculation has to be done on the resulting point, to ensure it is within the maximum range.
The test, using the scenario mentioned above improved the lookup time from approximately 70ms to 2ms, which is an increase by factor 35.

5.3.5. API Reference

This section explains the most important API endpoints for the vector layer. A full list can be found in the appendix in section A.3.2. Note that features are represented as „IEnumerable<IGeoCoordinate>“. This makes understanding and constructing features easier for users of the API. The layer understands this representation and it makes the API independent from NTS.

```
IGeoCoordinate GetClosestPoint(GeoCoordinate gpsCoordinate,
double maxDistance);
```

Listing 5.12: Vector API – GetClosestPoint

This method returns the coordinate of the nearest neighbor with the specification of a max distance in km.

```
double Distance(int featureIndex, IEnumerable<IGeoCoordinate> coords);
double DistanceInKm(int featureIndex, IEnumerable<IGeoCoordinate> coords);
```

Listing 5.13: Vector API – Distance

This calculates the distance between two features in degree. Since the unit is not very intuitive, an overloaded function exists, which converts the result to kilometers.

```
bool IsMultiPointInside(IEnumerable<IGeoCoordinate> coords);
bool IsLineStringInside(IEnumerable<IGeoCoordinate> coords);
bool IsMultiPointCrossing(IEnumerable<IGeoCoordinate> coords);
bool IsLineStringCrossing(IEnumerable<IGeoCoordinate> coords);
bool IsMultiPointIntersecting(IEnumerable<IGeoCoordinate> coords);
bool IsLineStringIntersecting(IEnumerable<IGeoCoordinate> coords);
bool IsMultiPointOverlapping(IEnumerable<IGeoCoordinate> coords);
bool IsLineStringOverlapping(IEnumerable<IGeoCoordinate> coords);
```

Listing 5.14: Vector API – Correlations

The methods listed above allows to calculate a wide variety of correlations between features.

```
double GetAccumulatedPathRating(IGeoCoordinate source,
IGeoCoordinate target, int distance);
double GetAccumulatedPathRating(IGeoCoordinate source,
int distance, double bearing);
```

Listing 5.15: Vector API – Path Rating

The path rating is a special method designed to make movement in an obstacle environment easier. It takes a source coordinate and a distance, as well as either a target coordinate or a bearing in degree (0-359). It calculates, if the line between the points cross a feature. If that is the case, the crossed feature's data table is checked for a „resistance“ value. If such a value exist, it is the result of the method, otherwise the path rating is 0.

```
object GetTimeseriesDataForCurrentTick();
```

Listing 5.16: Vector API – Time-series Sata

This returns the time-series value for the current tick. The use of the type „object“ allows the user to store more complex values than simple numbers.

```
object GetFromDataTable(int featureId, string key);
void AddToDataTable(int featureId, string key, object value);
```

Listing 5.17: Vector API – Data Table

The methods above allow reading and writing from a feature's data table. The use of the „object“ type has the same reason as above.

6. Experiments

The experiments chapter covers a practical test with minimal models. It also contains a performance test that compares the new layers to the previous one, to show that the GIS implementation does not sacrifice the simulation's performance.

6.1. Experimental Setup

The experiments consist of the four main use-cases, which are described in section 3.2. They are obstacle detection, sensing, time-series and writing data to the raster.
Each use-case has three models, one for each type, the old implementation, the GIS Raster Layer and the GIS Vector Layer. The only exception is the write data use-case. Since it is not supported by any of the old layers, there will be no test, which leads to 11 tests in total.
For each test, a model was created. The models are a minimal implementation using one layer and 15,000 agents with latitude, longitude and a reference to one of the layer types. For each tick, every agent executes an action depending on the use-case.

The simulation were done in the production cluster with LIFE version 2.5.9. This environment contains of several notes. Each note was run on an Apple Mac Pro (2013) with a 6 core (12 including hyper-threading) Intel Xeon processor and 64 GB memory.
The fact that the server cluster is used by others can lead to varying results. In order to reduce this factor, the simulation have been run three times. To eliminate peaks hours, the simulation were done at different times over 2 days.
Every model was simulated for 31 days with a tick duration of one hour, resulting in 744 total ticks. The total number of simulation sums up to 33 with 24,552 simulated ticks.
The result output, which takes approximately one second per tick was disabled to increase accuracy of the results. Measurements were taken from the simulations console output that logs the execution time of every tick.

6.2. Execution

All models were created in one ASP.NET Core solution and the code can be found in a project called „thesis-experiments“ inside the MARS GitLab. This contains all the compiled models, as well a the input files.
The execution of each test was done by uploading model and data to the WebUI. The data was then mapped to the model and executed. The console log shows the execution time of each tick.

6.2.1. Obstacle

The obstacle use-case is meant to simulate agent movement withing boundaries. During each tick an agent calculates a position from a random distance between 20 meters and 7 kilometers and a random direction between 0 and 359 degrees.
The path from the current to the new location is then checked for interfering obstacles.

6.2.2. Sensing

Sensing expresses the agents capabilities to detect an area or object of interest and move towards it. It does not matter if sensing is done by line of sight, smelling or other methods of detecting the target.
In case of the GeoPotential Field layer and the GIS Raster Layer, it requires an already initiated file with increasing values towards the target. The layers check for the surrounding cells and move towards the highest, until reaching the maximum is reached. The GIS Vector Layer does a nearest neighbor lookup within a specific radius.

6.2.3. Time-series

Both the old time-series layer and the GIS Vector layer can only store one value for a certain point of time. The Raster layer allows to store one file for a point in time.
The Layer replaces it’s current value automatically, depending on the simulation time. This guarantees that a requested value is always the current one.

6.2.4. Write to Raster

Writing to the raster file in parallel allows agents to add values to the raster or subtract them. Setting to a specific value is not supported, since it would interfere with the parallelism.
In the test, each agent increments the value of the cell at his position by one on every tick.

6.3. Results

The results show the execution time of the tests as described before.

6.3.1. Obstacle

The performance metrics in figure 6.1 show that the original Obstacle Layer performs the best. It consistently provides tick times of about 11ms less than the GIS Raster implementation, which works in a similar way. This adds up to 8 seconds of additional simulation time per month.
However 11ms are not something that will be noticeable in a real environment, where approximately 1,000ms of time is added for persisting results and other operations.
The Vector layer offers performance values that are in the middle, making it a slightly better alternative to the GIS Raster Layer.

Overall, the performance of the three layers are in the same range. This being said, the type of layer being used, will be defined by the data, the user has available and not the layer performance.

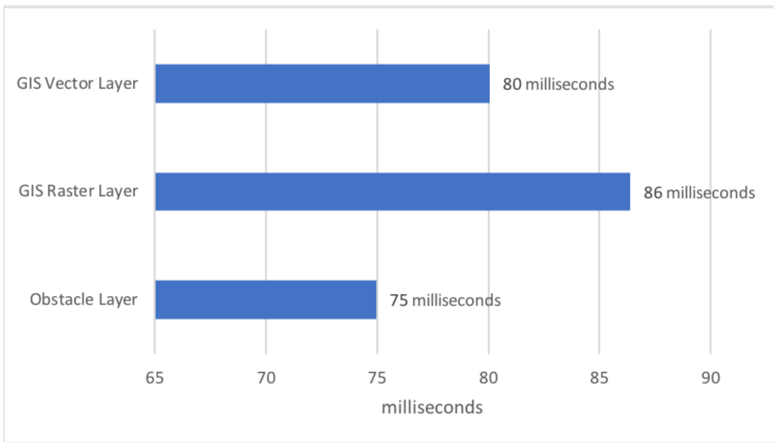

Figure 6.1.: Obstacle performance

6.3.2. Sensing

The sensing performance values, as seen in figure 6.2 show much diverse results than the previous test.
As expected, the nearest neighbor detection of the GIS Vector Layer does not perform as good, as the implementation of a raster file that was specially created for this use-case.

However, the bad performance of the Potential Field Layer is surprising, since it's implementation relies on a raster file, just like the GIS Raster Layer.

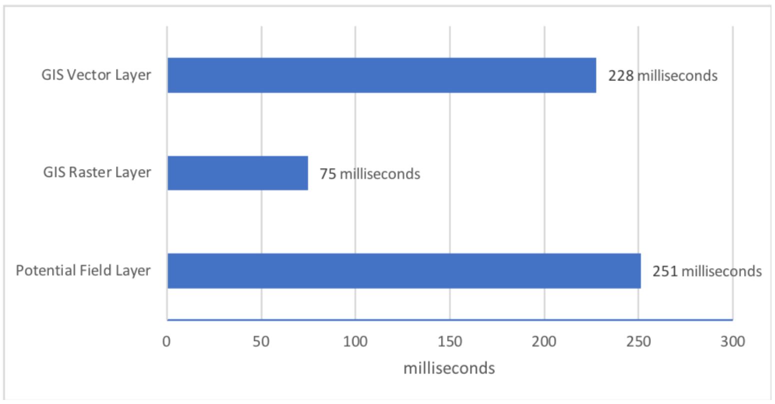

Figure 6.2.: Potential Field performance

6.3.3. Time-series

The Timeseries Layer and the the GIS Vector Layer's time-series implementation both rely on a single value, explaining the comparable performances. The GIS Raster Layer has to reinitialize the input file that is being used, depending on the current tick. Therefore it is slightly slower than the other implementations shown in figure 6.3.

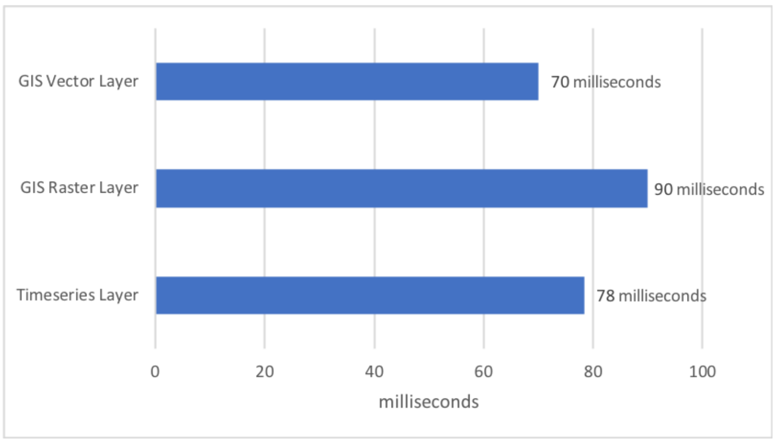

Figure 6.3.: Timeseries performance

6.3.4. Write to Raster

The write performance as shown in figure 6.4 does not include a previous layer, as such a layer does not exist.
Writing data in case of the GIS Vector Layer changes the files data table and in case of the GIS Raster Layer the cell values of the file. The GIS Raster Layer performs better. This is the case, because the raster agents cannot simply increment a value, but have to read and parse the current value before setting a new one.

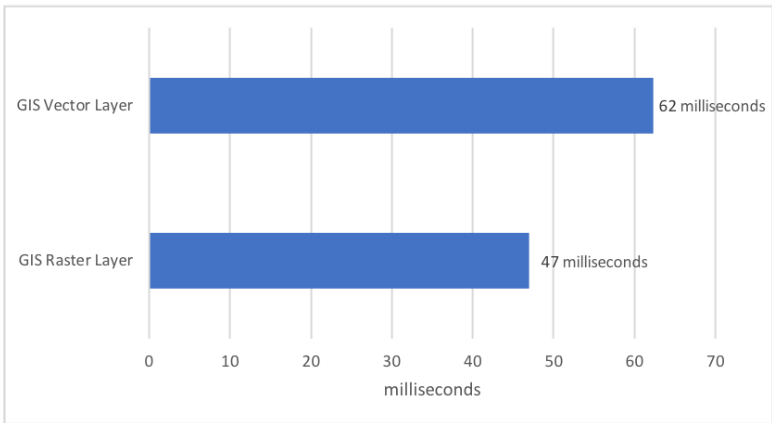

Figure 6.4.: Write Data performance

7. Conclusion & Future Work

This chapter concludes the thesis and gives an outlook of future work.

7.1. Conclusion

This thesis introduced two data layers to the MARS simulation that allow model developers to take advantage of GIS based input data files. The use-case specific layers, together with their custom file formats and specialized converters are therefore obsolete.
The new layers were integrated into the KNP model which is the biggest model for the MARS system. Also the DSL created by Glake (2018) makes use of the GIS Raster Layer, allowing users without advanced knowledge in programming to create simulations using GIS. The implementation of GIS Vector support is currently planned. With this being said, the work is already a central part of the MARS ecosystem.
The validity of the hypotheses, defined in section 1.1 will be discussed in the following section.

7.1.1. Hypotheses Validation

H1 – Introduce GIS and retain existing Functionality

GIS was successfully added to the LIFE simulation system and is capable of fulfilling all the desired use-cases. Those include both the previous features and the writing of data to raster layers during runtime. The implementation also improves upon previous features. E.g. detection of the closest cell, supports fixed value and function parameters to discover cells in ranges of values. This hypothesis is therefore validated.

H2 – Create a generic Importer

A uniform importer that supports both types of GIS was created. It automatically detects the GIS type and is capable of converting input data to the formats, required by the GIS layers. Hypothesis 2 is therefore valid.

H3 – Make custom Generators obsolete

The native support for GIS as GeoJSON, Shapefile, AsciiGrid and GeoTIFF is supported. This Allows direct use of the mentioned file types. However, the support for time-series data requires to provide additional files, in a form that is not supported by default GIS applications. The files do not alter the original input and consist of standardized CSV files which don't require special processing. Therefore it can be concluded that hypothesis 3 is still valid.

H4 – GIS does no diminish the Performance

The metrics in section 6.3 show varying results. The obstacle performance could not be reached by either of the GIS layers. The Raster Layer performed worst, results being 12.79% (11ms) slower than the obstacle layer.
The sensing of the Potential Field Layer and the GIS Vector Layer were about the same, but the GIS solution showed slightly better performance and the GIS Raster Layer being substantially faster.
The time-series performance was about the same, with the Vector layer slightly ahead and the Raster Layer being the slowest.
Overall performance of the GIS implementation was better, but due to the obstacle performance, the validity of this hypothesis is debatable.

7.2. Future Work

Although the layers that were introduced in this work, function as desired, they could be further improvement. The GIS Raster time-series implementation uses a metadata file to associate input files with the current point in time. In the future it might be required to allow multiple files for a certain period. This could be useful, if one input file has a higher accuracy than another in certain conditions.
Currently the GIS Raster Layer supports sensing by finding the highest adjacent cell. This could be extended by accepting a function as a parameter which would allow route finding between grid cells, based on a predicate logic (e.g. routing through a mountain area with minimal elevation change).
The initialization speed could be improved in the future, by implementing readers for binary input files. The GIS Raster layer could then be initialized by a GeoTIFF file and the GIS Vector Layer by using an Esri Shapefile. This is particular valuable for the GIS Raster Layer, because it's files tend to be larger and therefore take an increased amount of time to initialize.

The performance of the obstacle detection could potentially be optimized. This would involve introducing more advanced algorithms. Leveraging the power of Graphics processing unit (GPU)s might also be a vital option.

A. Appendix

A.1. Acknowledgment

I would like to thank Prof. Dr. Thomas Clemen for his valuable input and support during the time of my master studies. Without you, the casual and productive work environment of the MARS group, would not exist.
My thank also goes to Julius Weyl, Daniel Glake, Petar Krastev, Prannoy Mulmi and the rest of the MARS Team. You technical input and constructive criticism is highly appreciated. It was both enriching and fun working with you all.
I would also like to thank Prof. Dr. Stefan Sarstedt in his role as secondary examiner.
Last, but not least my thank goes to my parents. Without your moral and financial support my studies would not have been possible way.

A.2. Acronyms

API	Application programming Interface
CRS	Coordinate Reference System
GIS	Geographic Information Systems
GPS	Global Positioning System
MARS	Multi-Agent Research & Simulation
MAS	Multi-Agent Simulation
WGS 84	World Geodetic System 1984
EPSG	European Petroleum Survey Group
EGM2008	Earth Gravitational Model 2008
NGA	National Geospatial-Intelligence Agency

GeoTIFF	Georeferenced Tagged Image File Format
JSON	JavaScript Object Notation
GDAL	Geospatial Data Abstraction Library
OSGEO	The Open Source Geospatial Foundation
OGR	OpenGIS Simple Features Reference Implementation
NTS	NetTopologySuite
OGC	Open Geospatial Consortium
IETF	Internet Engineering Task Force
DSL	Domain Specific Language
KNP	Kruger National Park
WFS	Web Feature Service
WCS	Web Coverage Service
BST	Binary search tree
REST	Representational State Transfer
JAR	Java Archive
GPU	Graphics processing unit

A.3. API Documentation

A.3.1. GIS Raster Layer

```
/// <summary>
/// The structure storing the grid data.
/// </summary>
IReadOnlyDictionary<IGridCoordinate, double?> Grid { get; }

/// <summary>
/// The metadata of the layer.
/// </summary>
IAscMetadata Metadata { get; }

/// <summary>
/// Adds a value to an existing grid cell.
/// If the Cell does not exist, an ArgumentOutOfRangeException
/// is thrown.
/// </summary>
/// <param name="coord">The grid coordinate</param>
/// <param name="value">The value you are adding.</param>
void AddToGridField(IGridCoordinate coord, double value);

/// <summary>
/// Adds a value to an existing grid cell on the gps position.
/// If the Cell does not exist, an ArgumentOutOfRangeException
/// is thrown.
/// </summary>
/// <param name="gps">The gps coordinate</param>
/// <param name="value">The value you are adding.</param>
void AddToGridField(IGeoCoordinate gps, double value);

/// <summary>
/// Substracts a value from an existing grid cell.
/// If the Cell does not exist, an ArgumentOutOfRangeException
/// is thrown.
/// </summary>
/// <param name="coord">The grid coordinate</param>
/// <param name="value">The value you are substracting.</param>
void SubtractFromField(IGridCoordinate coord, double value);
```

```

/// <summary>
/// Substracts a value from an existing grid cell on the gps position.
/// If the Cell does not exist, an ArgumentOutOfRangeException
/// is thrown.
/// </summary>
/// <param name="gps">The gps coordinate</param>
/// <param name="value">The value you are substracting.</param>
void SubtractFromField(IGeoCoordinate gps, double value);

/// <summary>
/// Try to reduce a cell rating by a given value
/// </summary>
/// <param name="coord">The grid coordinate</param>
/// <param name="amountToTake">value by which the cell rating should
/// be reduced</param>
/// <returns>the amount that was took or 0 if there isn't
/// enough left</returns>
double TryToSubtractFromField(IGridCoordinate coord,
    double amountToTake);

/// <summary>
/// Try to reduce a cell rating by a given value
/// </summary>
/// <param name="position">gps position to find the cell</param>
/// <param name="amountToTake">value by which the cell rating should
/// be reduced</param>
/// <returns>The amount that was took or 0 if there isn't
/// any left</returns>
double TryToSubtractFromField(IGeoCoordinate position,
    double amountToTake);

/// <summary>
/// Retrieves the value from a pixel at a position.
/// </summary>
/// <param name="coord">pixel coordinate.</param>
/// <returns></returns>
double GetValue(IGridCoordinate coord);

/// <summary>
/// Retrieves the value from a GPS position.
/// </summary>
/// <param name="gps">GPS position of the the target
/// location.</param>
/// <returns>Target value.</returns>
double GetValue(IGeoCoordinate gps);

/// <summary>
/// Get the coordinate positon of the the adjacent cell
/// with the highest value
/// </summary>
/// <param name="coord">pixel coordinate.</param>
/// <returns>position of the cell with the highest value</returns>
IGridCoordinate GetNeighbourCellWithMaxValue(IGridCoordinate coord);

/// <summary>
/// Get the gps positon of the the adjacent cell with the
/// highest value
/// </summary>
/// <param name="positon">gps position where the
/// search starts</param>
/// <returns>position of the cell with the highest value</returns>
IGeoCoordinate GetNeighbourCellWithMaxValue(IGeoCoordinate positon);

/// <summary>
/// Returns the closest cell with the specified value
/// </summary>
/// <param name="coord">Initial cell</param>
/// <param name="value">Desired value</param>
/// <param name="maxDistance">The maximum distance of cells
/// to search</param>
/// <returns></returns>
IGridCoordinate GetClosestCellWithValue(IGridCoordinate coord,
    double value, int maxDistance);

/// <summary>
/// Returns the closest cell coordinate with the specified value
/// </summary>
/// <param name="gps">Initial cell coordinate</param>
/// <param name="value">Desired value</param>
/// <param name="maxDistance">the maximum distance of cells
/// to search for</param>
/// <returns></returns>
IGeoCoordinate GetClosestCellWithValue(IGeoCoordinate gps,
    double value, int maxDistance);

/// <summary>
/// Returns the closest cell with the specified predicat
/// </summary>
/// <param name="coord">Initial cell</param>
/// <param name="predicat">Expression that is checked to find
/// the cell</param>
/// <param name="maxDistance">the maximum distance of cells
/// to search</param>
/// <returns></returns>
IGridCoordinate GetClosestCellWithValue(IGridCoordinate coord,
Func<double, bool> predicat, int maxDistance);

/// <summary>
/// Returns the closest cell coordinate with the specified value
/// </summary>
/// <param name="cell">Initial cell coordinate</param>
/// <param name="value">Desired value</param>
/// <param name="maxDistance">the maximum distance of cells
/// to search</param>
/// <returns></returns>
IGeoCoordinate GetClosestCellWithValue(IGeoCoordinate cell,
    double value, int maxDistance);


/// <summary>
/// Retrieves all data.
/// </summary>
/// <returns>Metadata in string representation.</returns>
string PrettyPrintGrid();
```

Listing A.1: GIS Raster Layer API

Metadata

```
int WidthInGridCells { get; }

int HeightInGridCells { get; }

IGeoCoordinate UpperRightBound { get; }

IGeoCoordinate LowerLeftBound { get; }

double CellSizeInDegree { get; }

int NoDataValue { get; }

bool IsInsideBounds(IGridCoordinate coord);

bool IsInsideBounds(IGeoCoordinate gps);

string PrettyPrintMetadata();
```

Listing A.2: Raster Layer Metadata API

A.3.2. GIS Vector Layer

```
/// <summary>
/// The complete dataTable accessible
/// </summary>
ConcurrentDictionary<string, object>[] DataTable { get; }

/// <summary>
/// The complete TimeseriesData accessible
/// </summary>
SortedList<DateTime, double> TimeseriesData{ get; }

/// <summary>
/// Gets the current Index of timeseries file
/// </summary>
int CurrentTsIndex { get; }

/// <summary>
/// Writes the DataTable back to the original structure and returns
/// it as JSON. This opperation is not threadsafe, so don’t call
/// this function in paralles, e.g. in the agent logic. It is meant
/// to be used for result output between ticks.
/// </summary>
/// <returns>All features with datatable in JSON</returns>
string GetFeaturesAsJson();

/// <summary>
/// Gets a feature at a certain index.
/// </summary>
/// <param name="featureIndex">Index of the requested feature</param>
/// <returns>The feature</returns>
IEnumerable<IGeoCoordinate> GetFeature(int featureIndex);

/// <summary>
/// Gets the closest point to a GPS position from all features.
/// </summary>
/// <param name="gpsCoordinate">GPS coordinate</param>
/// <param name="maxDistance">Distance in KM</param>
/// <returns>The feature</returns>
IGeoCoordinate GetClosestPoint(GeoCoordinate gpsCoordinate,
    double maxDistance);

/// <summary>
/// Calculates the closest distace between a feature and
/// other features.
/// </summary>
/// <param name="featureIndex">Index of the source feature.</param>
/// <param name="coords">List of target features.</param>
/// <returns>Distance to the closest feature in Degree.</returns>
double Distance(int featureIndex,
    IEnumerable<IGeoCoordinate> coords);

/// <summary>
/// Calculates the closest distace between a feature and
/// other features.
/// </summary>
/// <param name="featureIndex">Index of the source feature.</param>
/// <param name="coords">List of target features.</param>
/// <returns>Distance to the closest feature in KM.</returns>
double DistanceInKm(int featureIndex,
    IEnumerable<IGeoCoordinate> coords);

```

```
/// <summary>
/// Determines if a point is inside the current feature.
/// </summary>
/// <param name="coord">A coordinate that is interpreted
/// as a unique point.</param>
/// <returns>True, if the point is inside.</returns>
bool IsPointInside(IGeoCoordinate coord);

/// <summary>
/// Determines if points are inside the current feature.
/// </summary>
/// <param name="coords">List of coordinates that are interpreted
/// as unique points.</param>
/// <returns>True, if any point is inside.</returns>
bool IsMultiPointInside(IEnumerable<IGeoCoordinate> coords);

/// <summary>
/// Determines if a LineString are inside the current feature.
/// </summary>
/// <param name="coords">List of coordinates that are interpreted
/// as LineString</param>
/// <returns>True, if any point is inside.</returns>
bool IsLineStringInside(IEnumerable<IGeoCoordinate> coords);

/// <summary>
/// Determines if coords are crossing the current feature.
/// </summary>
/// <param name="coords">List of coordinates that are interpreted
/// as MultiPoint.</param>
/// <returns>True, if any two points are crossing.</returns>
bool IsMultiPointCrossing(IEnumerable<IGeoCoordinate> coords);

/// <summary>
/// Determines if coords are crossing the current feature.
/// </summary>
/// <param name="coords">List of coordinates that are interpreted
/// as LineString.</param>
/// <returns>True, if any two strings are crossing.</returns>
bool IsLineStringCrossing(IEnumerable<IGeoCoordinate> coords);

```

```
/// <summary>
/// Determines if coords are intersect the current feature.
/// </summary>
/// <param name="coords">List of coordinates that are interpreted
/// as MultiPoint.</param>
/// <returns>True, if any two points are intersection.</returns>
bool IsMultiPointIntersecting(IEnumerable<IGeoCoordinate> coords);

/// <summary>
/// Determines if coords are intersect the current feature.
/// </summary>
/// <param name="coords">List of coordinates that are interpreted
/// as LineString.</param>
/// <returns>True, if any two lines are intersection.</returns>
bool IsLineStringIntersecting(IEnumerable<IGeoCoordinate> coords);

/// <summary>
/// Determines if coords are overlaping the current feature.
/// </summary>
/// <param name="coords">List of coordinates that are interpreted
/// as MultiPoint.</param>
/// <returns>True, if any two points are overlapping.</returns>
bool IsMultiPointOverlapping(IEnumerable<IGeoCoordinate> coords);

/// <summary>
/// Determines if coords are overlaping the current feature.
/// </summary>
/// <param name="coords">List of coordinates that are interpreted
/// as LineString.</param>
/// <returns>True, if any two lines are overlapping.</returns>
bool IsLineStringOverlapping(IEnumerable<IGeoCoordinate> coords);

/// <summary>
/// Calculates the path rating for a given path. This checks if any
/// paths are crossing. If that is the case, the "resistance" value
/// is read from the features data table. These values are added up.
/// </summary>
/// <param name="source">Source position</param>
/// <param name="target">Target position</param>
/// <param name="distance">The speed.</param>
/// <returns>the pathrating(resistance) of the path</returns>
double GetAccumulatedPathRating(IGeoCoordinate source,
    IGeoCoordinate target, int distance);

/// <summary>
/// Calculates the path rating for a given path. This checks if any
/// paths are crossing. If that is the case, the "resistance" value
/// is read from the features data table. These values are added up.
/// </summary>
/// <param name="source">The starting point.</param>
/// <param name="distance">The speed.</param>
/// <param name="bearing">The front facing direction messured
/// in degree.</param>
/// <returns>the pathrating(resistance) of the path.</returns>
double GetAccumulatedPathRating(IGeoCoordinate source,
    int distance, double bearing);

/// <summary>
/// Returns the timeseries data for the current tick if available.
/// </summary>
object GetTimeseriesDataForCurrentTick();

/// <summary>
/// Reads data from the file’s data table.
/// Note: for performance and concurrency this is just done in
/// the memory representation and not in the file.
/// </summary>
/// <param name="featureId">Feature to be queried.</param>
/// <param name="key">Key to be searched.</param>
/// <returns>The value object of the request.</returns>
object GetFromDataTable(int featureId, string key);

/// <summary>
/// Write data to the file’s data table.
/// Note: for performance and concurrency this is just done in
/// the memory representation and not in the file.
/// </summary>
/// <param name="featureId">Feature to be queried.</param>
/// <param name="key">Key to be inserted.</param>
/// <param name="value">Value to be inserted.</param>
/// <returns>The value object of the request.</returns>
void AddToDataTable(int featureId, string key, object value);
```

Listing A.3: GIS Vector API

Bibliography

arcgis.com (2018). *Raster feature cell size.* [Image Resource]. URL: http://desktop.arcgis.com/en/arcmap/latest/manage-data/raster-and-images/what-is-raster-data.htm (visited on 07/20/2018).

Bentley, Jon Louis (1975). „Multidimensional Binary Search Trees Used for Associative Searching“. In: *Commun. ACM* 18.9, pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. (Visited on 07/19/2018).

Bledsoe, Brian P, Michael C Brown, and David A Raff (2007). „GeoTools: A Toolkit for Fluvial System Analysis 1“. In: *JAWRA Journal of the American Water Resources Association* 43.3, pp. 757–772. DOI: 10.1111/j.1752-1688.2007.00060.x. (Visited on 07/19/2018).

Butler, Howard et al. (2008). „The GeoJSON format specification“. In: *Rapport technique* 67. URL: http://geojson.org/geojson-spec (visited on 07/19/2018).

Butler, H. et al. (2016). *The GeoJSON Format.* Tech. rep. DOI: 10.17487/RFC7946. (Visited on 07/19/2018).

Cetin, Mehmet et al. (2018). „Mapping of bioclimatic comfort for potential planning using GIS in Aydin“. In: *Environment, Development and Sustainability* 20.1, pp. 361–375. URL: https://link.springer.com/article/10.1007/s10668-016-9885-5 (visited on 07/20/2018).

Consortium, Open Geospatial et al. (2010). „OpenGIS Implementation Standard for Geographic information-Simple feature access-Part 2: SQL option“. In: *by Herring, JR* 1.1, pp. 15–17. URL: http://www.opengeospatial.org/standards/sfs (visited on 07/19/2018).

Consortium, Open Geospatial (2014). *OGC® GeoPackage Encoding Standard.* URL: http://www.opengeospatial.org/standards/geopackage (visited on 07/19/2018).

Decker, B LOUIS (1986). *World geodetic system 1984.* Tech. rep. Defense Mapping Agency Aerospace Center St Louis Afs Mo, pp. 21–23. URL: http://www.dtic.mil/docs/citations/ADA167570 (visited on 07/19/2018).

ESRI, ESRI (1998). „Shapefile technical description“. In: *An ESRI White Paper.*

Ghorbani Nejad, Samira et al. (2017). „Delineation of groundwater potential zones using remote sensing and GIS-based data-driven models“. In: *Geocarto international* 32.2, pp. 167–187. DOI: 10.1080/10106049.2015.1132481. (Visited on 07/20/2018).

Giles-Corti, Billie et al. (2016). „City planning and population health: a global challenge“. In: *The lancet* 388.10062, pp. 2912–2924. DOI: 10.1016/S0140-6736(16)30066-6. (Visited on 07/19/2018).

Glake, Daniel (2018). „MARS DSL: Eine typisierte Sprache zur Modellierung komplexer agentenbasierter Modelle“. MA thesis. Hamburg University of Applied Sciences - HAW Hamburg.

Grafarend, Erik W (1995). „The optimal universal transverse Mercator projection“. In: *Manuscr. Geodaet.* Vol. 20. 6. Springer, pp. 421–468. DOI: 10.1007/978-3-642-79824-5_13. (Visited on 07/19/2018).

Hüning, Christian (2016). „Analysis of Performance and Scalability of the Cloud-Based Multi-Agent System MARS“. MA thesis. Hamburg University of Applied Sciences - HAW Hamburg. URL: http://edoc.sub.uni-hamburg.de/haw/volltexte/2017/3722/ (visited on 07/19/2018).

Hüning, Christian, Mitja Adebahr, et al. (2016). „Modeling & simulation as a service with the massive multi-agent system MARS“. In: p. 1. URL: https://dl.acm.org/citation.cfm?id=2972194 (visited on 07/19/2018).

Hüning, Christian, Jason Wilmans, et al. (2014). „MARS-A next-gen multi-agent simulation framework“. In: pp. 1–14. URL: https://mars-group.org/wp-content/uploads/papers/MARS%20-%20A%20next-gen%20multi-agent%20simulation%20framework.pdf (visited on 07/19/2018).

Karsten, Lennart (2018). *Optimizing geospatial read-performance inside a multi-agent simulation system.*

Maguire, David J (1991). „An overview and definition of GIS“. In: *Geographical information systems: Principles and applications* 1, pp. 9–20.

mapserver.org (2018). *Grid Displaying Degrees with Symbol.* [Image Resource]. URL: http://mapserver.org/_images/grid-degrees-symbol.png (visited on 07/20/2018).

Meer (2012). *Atlas of the World. Gerard Mercator's Map of the World (1569).* Walburg Pers bv. URL: https://www.karwansaraypublishers.com/mercator-atlas.html (visited on 07/19/2018).

Momjian, Bruce (2001). *PostgreSQL: introduction and concepts.* Vol. 192. Addison-Wesley New York. URL: www.foo.be/docs-free/aw_pgsql_book.pdf (visited on 07/19/2018).

National Geospatial Intelligence Agency (2014). *Implementation Practice - Web Mercator Map Projection*. Tech. rep. National Geospatial-Intelligence Agency (NGA), p. 15. URL: http://earth-info.nga.mil/GandG/wgs84/web_mercator/index.html (visited on 07/19/2018).

Obe, Regina and Leo Hsu (2011). „PostGIS in action“. In: *GEOInformatics* 14.8, p. 30. URL: https://www.manning.com/books/postgis-in-action-second-edition (visited on 07/19/2018).

Pavlis, Nikolaos K et al. (2012). „The development and evaluation of the Earth Gravitational Model 2008 (EGM2008)“. In: *Journal of geophysical research: solid earth* 117.B4. DOI: 10.1029/2011JB008916. (Visited on 07/19/2018).

Pinto, Domingos et al. (2017). „Delineation of groundwater potential zones in the Comoro watershed, Timor Leste using GIS, remote sensing and analytic hierarchy process (AHP) technique“. In: *Applied Water Science* 7.1, pp. 503–519. DOI: 10.1007/s13201-015-0270-6. (Visited on 07/20/2018).

qgis.org (2018). *The three families of map projections*. [Image Resource]. URL: https://docs.qgis.org/testing/en/_images/projection_families.png (visited on 07/20/2018).

Ritter, N. and M. Ruth (1997). „The GeoTIFF data interchange standard for raster geographic images“. In: *International Journal of Remote Sensing* 18.7, pp. 1637–1647. ISSN: 13665901. DOI: 10.1080/014311697218340. (Visited on 07/19/2018).

Ritter, Niles et al. (2000). „GeoTIFF format specification GeoTIFF revision 1.0“. In: *SPOT Image Corp.*

Růžička, Jan (2016). „Comparing speed of Web Map Service with GeoServer on ESRI Shapefile and PostGIS“. In: *Geoinformatics FCE CTU* 15.1, pp. 3–9. DOI: 10.14311/gi.15.1.1. (Visited on 07/20/2018).

Schwartz, Joe (2018). *Bing Maps Tile System*. [Image Resource]. URL: https://msdn.microsoft.com/en-us/library/bb259689.aspx (visited on 07/20/2018).

Strebe, Daniel R. (2011). *Mercator projection of the world between 82°S and 82°N*. [Image Resource]. URL: https://en.wikipedia.org/wiki/Mercator_projection%5C#/media/File:Mercator_projection_SW.jpg (visited on 07/20/2018).

Toews, M. W. (2007). *Simple vector map*. [Image Resource]. URL: https://commons.wikimedia.org/w/index.php?curid=3024482 (visited on 07/20/2018).

Toups, Matthew A (2016). „A study of three paradigms for storing geospatial data: distributed-cloud model, relational database, and indexed flat file“. In: URL: http://scholarworks.uno.edu/td/2196 (visited on 07/20/2018).

Wang, Zhiyong and Sisi Zlatanova (2016). „Multi-agent based path planning for first responders among moving obstacles“. In: *Computers, Environment and Urban Systems* 56, pp. 48–58. DOI: 10.1016/j.compenvurbsys.2015.11.001. (Visited on 07/20/2018).

Wooldridge, Michael (2009). *An introduction to multiagent systems*. John Wiley & Sons. URL: https://books.google.de/books?hl=en&lr=&id=X3ZQ7yeDn2IC (visited on 07/19/2018).

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

Hamburg, July 26, 2018 Lennart Karsten