UNIVERSITY OF TARTU

Institute of Computer Science

Software Engineering Curriculum

Suido Valli

Visualization of simulations of a robot operated car park system

Master’s Thesis (30 ECTS)

Supervisor(s): Dirk Oliver Theis

Tartu 2016

Visualization of simulations of a robot operated car park system

Abstract:

Due to the increasing number of vehicles on the road, traffic problems are bound to exist.This happens as the current car park facilities developed are unable to cope with the influxof vehicles on the road. To overcome this problem, robot operated car park system is developed.This solution would use already built parking lots and parking houses, but will allowsaid parking facilities to accommodate more vehicles as the size of the parking spot will bereduced and most of the roads leading to the parking spots can be eliminated. Other solutionsthat exist will need specially built parking facilities, which makes them more costly to implement.The aim of this thesis is to come up with the best solution for visualizing the algorithmused in the robot operated car park systems. As the algorithm behind it is complex, itis difficult to grasp it without proper visual presentation. The solutions found in the thesiswill be used in making a web application that will let the users interact with the system toget high-level overview of the algorithm’s inner workings

Keywords:

Interfacing, web application, simulation, visualization

**CERCS:**

P170, Computer science, numerical analysis, systems, control

Lühikokkuvõte:

Selle lõputöö tulem on robotiseeritud autoparklate süsteemi simulatsioone visualiseeriv veebiaplikatsioon. Meie ümber on üha rohkem autosid ning metropolides rohkem inimesi. Autode üleküllus viib parkimiskohtade puuduseni. Selle üheks lahenduseks sobiks seniste parklate ümbermuutmine robotiseeritud parkimissüsteemideks – see hoiaks kokku nii maaala kui ka raha, mis täiesti uute parkimiskonstruktsioonide ehitamise jaoks vaja läheks. Selle töö tulemuseks on veebirakendus mis visualiseerib sellise parkimissüsteemi taga oleva algoritmi tööd arusaadavalt ja meeldivalt

Võtmesõnad:

Integratsioon, veebirakendus, simulatsioon, visualiseerimine

**CERCS:**

P170, Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

[1 Introduction 5](#_Toc450937867)

[2 Overview 6](#_Toc450937868)

[2.1 Robot car park system 6](#_Toc450937869)

[2.2 Similar solutions 7](#_Toc450937870)

[PathFinding.js 8](#_Toc450937871)

[VisuAlgo 8](#_Toc450937872)

[3 Application requirements 10](#_Toc450937873)

[3.1 Non-functional requirements 10](#_Toc450937874)

[Appearance 10](#_Toc450937875)

[Scalability 10](#_Toc450937876)

[Usability 10](#_Toc450937877)

[Reliability 10](#_Toc450937878)

[Correctness 10](#_Toc450937879)

[Durability 10](#_Toc450937880)

[Efficiency 10](#_Toc450937881)

[Reusability 10](#_Toc450937882)

[Extendibility 10](#_Toc450937883)

[Portability 11](#_Toc450937884)

[Documentation 11](#_Toc450937885)

[Installation 11](#_Toc450937886)

[3.2 Functional requirements 11](#_Toc450937887)

[Application is able to read the robroute file format 11](#_Toc450937888)

[Application is able to construct parking lot and machines’ instructions from a robroute instruction file 12](#_Toc450937889)

[Application is showing static images of start and end state of the simulation 12](#_Toc450937890)

[Application is visualizing the simulation of the algorithm 12](#_Toc450937891)

[Application is able to scale accordingly to parking lot and viewport size 12](#_Toc450937892)

[User is able to choose the parking lot layout to visualize from application 13](#_Toc450937893)

[4 System architecture 14](#_Toc450937894)

[4.1 Architecture overview 14](#_Toc450937895)

[4.2 Server side platform selection 14](#_Toc450937896)

[Python web framework selection 15](#_Toc450937897)

[4.3 Data object transmitting standards 16](#_Toc450937898)

[4.4 Client side platform selection 16](#_Toc450937899)

[JavaScript Libraries 17](#_Toc450937900)

[5 Application overview 20](#_Toc450937901)

[5.1 Application flow 21](#_Toc450937902)

[Rendering the index page 21](#_Toc450937903)

[Choosing the parking lot and instructions 21](#_Toc450937904)

[Rendering the main workspace 25](#_Toc450937905)

[Visualizing the scenario 25](#_Toc450937906)

[5.2 Application functions 26](#_Toc450937907)

[6 Validation 27](#_Toc450937908)

[7 Future opportunities 28](#_Toc450937909)

[8 Conclusions 29](#_Toc450937910)

[9 References 30](#_Toc450937911)

[Appendix 31](#_Toc450937912)

[I. Glossary 31](#_Toc450937913)

[II. License 32](#_Toc450937914)

# Introduction

In this thesis a new web application is introduced to visualize the simulation of robot operated car park system of which algorithm is developed by Dirk Oliver Theis at University of Tartu. The aim is to come up with the best solution for visualizing the algorithm that makes the algorithm easy to grasp. Furthermore, the visualization has to fill the requirements set by the algorithm. The solutions found in the thesis are used in making the web application that will show the work of the algorithm.

The main obstacles to overcome this thesis is to find the right tools for making a visually appealing, interactive simulation of visualization of robot car parking systems. Furthermore, the right tools will have to be chosen to interface with the algorithm library and with the user. The tools are required to work efficiently together and support visualization process. In addition, the thesis tries to answer to the question how to visualize these kind of algorithms the best for interactivity and understandability. This thesis covers the research of finding the most efficient and understandable way to visualize the simulation of robotized car parking system.

In addition, it covers the extensive practical work during the development of the web application. This includes the client part of the application, part of server that interfaces with the client and finally, part of server that interfaces with algorithm library. As the library is stateless, the application also has to hold the state of the parking lot in its memory.

The findings of the thesis are demonstrated in developing the aforementioned visualization application.

# Overview

In the following chapter the background of the application is described and compared to the similar solutions. In addition, it is explained why the proposed solution fits the best for the needs of the project.

## Robot car park system

Nowadays parking a car is more and more challenging task – especially in metropolitan areas. The problem is that the cities are getting denser, there are more people in the centre of the cities than there has ever been – but the area for parking spaces is not changing correlatively.

To overcome that problem, cars have to be fitted to the parking lot more densely than it is possible in regular parking lots. There are options in the market to do that, for example Ferris wheel-like parking systems, also known as paternoster systems [1]. These systems have been around from 1920’s and are starting to make now comeback in Europe and USA. The problem of paternoster system is the high cost of construction and need to overlook and usually demolish the existing infrastructure.

The other problem that can be seen usually in bigger or multi-story car parking lots is the time it takes to park and later to retrieve the car. It can take very long time, especially when the parking lot is operating almost in full capacity. Finding a place can be really arduous.

Newer parking lots and parking houses have got smarter. They have sensors that can see if the parking space is occupied or not. This information can be shown as aggregated data before entering the parking lot or parking house by how many free spaces, if any, are left. In addition, it is possible to show with green or red lights in front of every parking space if it is free or not.

Although these systems have made it parking the car more convenient, the capacity of a parking lot or parking house can be improved even more. So can the parking / retrieval time for the car owner be reduced dramatically.

This can be done by using an automated parking system that can use existing infrastructure with minimal changes needed. Suggested system (RCPS for short) is using a specified amount of small robots to park the cars.

The simplified flow of parking in RCPS parking lot from driver’s perspective is described:

* Driver drives the car to the designated marked area just outside the parking lot.
* Driver and all the passengers get out of the car.
* Driver locks the car and takes a ticket from the parking lot printer or uses a mobile application to park the car.
* Robot retrieves the car from the area driver left it.
* Robot takes the car into the parking lot and parks it.

As one can see, the process of a driver finding a parking space in a parking lot is eliminated from the parking process by RCPS.

The simplified flow of retrieving the car in RCPS parking lot from driver’s perspective is described:

* Driver inserts the ticket received from the parking lot printer to the parking lot information system or driver uses mobile application to send an order to the parking lot for car retrieval.
* Robot retrieves the car from parking lot and parks it outside the parking lot in designated retrieval area.
* Driver and passengers enter the car and drive away.

In conclusion, advantages of RCPS compared to usual parking lots are:

* Bigger car density
* No car damage (paint scratches by other people parking etc.)
* No exhaust fumes (robots are working with electrical motors)
* Dramatically reduced noise reduction (no combustion engines and people talking)
* Quicker car parking and retrieval for clients

These advantages, usually not all of them at once, are also the advantages of other automated parking systems compared to the usual parking lot. One of the main advantages compared to other automated parking systems of RCPS is the relative cheapness of it.

As said before, the basic infrastructure of the already existing parking lot or parking lot does not need to be demolished and can be used as a basis of RCPS. There is only need to install sensors, robot’s charging ports, new parking terminal and other relatively cheap things. Also the designated parking and retrieval spots have to be marked – they can take the place of entrance and exit of existing parking space. And, of course, the robots and installation has to be purchased.

The robots themselves look a lot like robotic vacuum cleaners – they are small and flat. Robots move cars by driving underneath one and then lifting the car up by that much that the tyres of the car do not touch the ground, move it to the parking space meant for the car and drop it.

The movement of cars and robots is controlled by the main controller, that gives instructions to the robots what to do at what timeframe. The algorithm for the said controller is being developed in University of Tartu and is part of an enterprise project ordered by external client.

To show the efficiency of the RCPS and algorithm beneath it, an application is to be made to visualize the effectiveness of RCPS compared to the usual parking lot.

## Similar solutions

The main objective the RCPS application tries to accomplish is to show potential customers that RCPS is more efficient with the parking space than the usual parking lot and the customers of parking lot will have better user experience.

The user experience consists of, amongst other indicators, the time it is needed to park the car and retrieve the car. These two indicators are directly correlated to how effectively the parking spaces in the parking lot are used. In busy days, there can be times that all of the spaces of a parking spaces are full. Now, the question is how does the RCPS handle these busy days and does it handle them better than people in usual parking lots would do that.

To present the RCPS to a customer, two different paths can be taken – the RCPS can be either presented on a big screen by the RCPS provider’s company to the customer, or customer can interact himself or herself with the virtual RCPS parking lot. The objective is to build an application that can be used for both options.

RCPS presentation is, in essence, nothing more than showing the movement of cars and robots in time.

In essence RCPS is an application that takes a parking lot as an input and will start executing movements on the screen relative of time. Also, it is visualizing the work of an algorithm. Taken those two approaches into account, similar solution can be found in algorithm visualization and time-based graphical movement. In some cases, they overlap.

### PathFinding.js

PathFinding.js itself is a path-finding library written in JavaScript for tile-based games[[1]](#footnote-1). The online demo of the library[[2]](#footnote-2) is for visualizing the different path finding algorithms implemented in the library. Upon opening the demonstration, client is introduced to a screen with a grid where all but two squares are white. The green square is the start position and red square the end position for path finding. User can add obstacles to the grid by using the mouse and move the start and end positions from their original positions. There is possibility to choose from 8 different algorithms and compare how they find the solution for the problem. When the user starts the visualization, every step of algorithm is shown on the screen with grey, green or blue squares, depending on the algorithm. When the algorithm has finished, user can see the length of the path, time it took and how many operations did the algorithm have to do.

The PathFinding.js demonstration is similar to RCPS in many ways – it visualizes the work of an algorithm, is somewhat interactive (user can give the input for the program) and is an application in Web browser. The differences are deal breakers – it is meant for demonstrating one type of algorithms only and is not capable of working with parking lots. Furthermore, the movement of the grids is not smooth – something that is a requirement to make the visualization of machine’s movement resemble real life situations as much as possible.

### VisuAlgo

VisuAlgo is an application that visualizes data structures and algorithms through animation[[3]](#footnote-3). User has a big choice of different data structures to choose from where each of the data structures has its own algorithms that can be visualized. The visualization can be shown automatically or step-by-step. In addition, pseudocode of the algorithm with highlighted step is shown on the screen for better understanding of inner workings of the algorithm. VisuAlgo has two modes, exploration mode where a user can simply discover the data structures and algorithms that go with them him/herself and e-Lecture mode, where the user will first learn about the data structure and has a tutorial of how to animate the algorithms before actually using them.

The overlapping of RCPS and VisuAlgo functionality is quite small – the only similarity is the abstract algorithm visualization. VisuAlgo is showing the algorithms quite differently from what RCPS has to. The purpose of VisuAlgo is to educate people on how different data structures and algorithm works. The aim of RCPS is to show that the algorithm used for car parking robots is more efficient than the contemporary manual parking system. The algorithm beneath it is a trade secret and the inner workings of it are not meant to be public.

# Application requirements

## Non-functional requirements

In the following subchapter, the non-functional requirements of the RCPS’s application are defined.

### Appearance

The application has to look appealing and pleasant to the customer.

### Scalability

The application can be used with up to 50 x 50 size parking lots and have instruction set of 1000.

Up to 200 users can use the application concurrently.

### Usability

The application can be used without extra manual by an English speaking person with medium computer usage skills.

### Reliability

The application works without any fatal problems 99% of the uptime.

### Correctness

The application shows the correct movement of the machines on the screen 99% of execution times.

### Durability

RCPS application will have an uptime of 23 hours and 30 minutes per day, 6 days, 12 hours per week and 28 days per month without any fatal crashes.

### Efficiency

The application must use less than 1 MB of network resources to show to load the parking lot with three different models of cars.

### Reusability

The application has to have the ability to be rewritten in minimal effort to visualize any other parking lot simulation with either implicit or explicit instruction set.

In case of implicit instruction set, an instruction set where the whole parking layout and every parking space’s state is written out for every instruction step. It is necessary for the RCPS application to convert these instructions to explicit instructions.

Explicit instruction set has explicit orders for every machine on the lot for every instruction step. For example, “R0”, “E” would mean that at that time, robot with ID R0 should move one step east.

### Extendibility

Event handler what takes the ID of machine, can be extended to be interactive simulation.

### Portability

The client application has to be usable from Linux, OS X and Windows operating systems when using Mozilla Firefox, Safari or Chrome web browsers. In addition, the client application has to be usable from Android and iOS mobile devices using respective native web browsers.

The server application has to run on Linux and Windows operating systems with Python 3.5+, bottle and Jinja2 installed.

### Documentation

The application has to have clear installation and usage instructions. The source code has to commit to the standards of the programming or mark-up language used and be thoroughly commented.

### Installation

The application can be installed to a server by a system administrator and run without any extra requirements than specified in the documentation. The client application does not need any installation.

## Functional requirements

Following requirements describe the functionality of the application to be created

* Application is able to read the robroute file format
* Application is able to construct parking lot and machines’ instruction from a robroute instruction file.
* Application is showing static images of the start and end state of the simulation
* Application is visualizing the simulation of the algorithm
* Application is able to scale accordingly to parking lot size and viewport size
* User is able to choose the parking lot layout from the application to visualize

In the following subchapters the requirements are explained in detail.

### Application is able to read the robroute file format

Robroute file format is the outcome of the robot car park system algorithm. It consists largely of four parts:

1. The width and height of the parking lot
2. The layout of parking lot
3. Number of movement steps
4. Movement steps

Width and height of the parking lot are two integers that describe will be the size of the parking lot.

The layout of the parking lot describes for each parking lot’s space to where it is possible to move from that parking space.

Number of movement steps shows how many movement instructions will follow in the file.

Movement steps show the state of every parking space in given step. Every parking space has four state variables.

1. onNode state shows what type of car is in the space at this step or if the space is empty.
2. ndStat shows the robot state in the space at the step. There can either be no robot or the robot can be ready, moving or lifting/dropping a car. Also, on the state it is shown either the robot is there with some type of car or alone.
3. rVertical state describes the vertical movement of the robot. It can either lift the car (there are 5 different levels of lifting), drop the car or there might be no vertical movement.
4. rMove state describes the horizontal movement of the robot. It describes the movement in all four possible directions (North, South, East or West), if the robot is moving with or without the car and is it accelerating or already moving.

### Application is able to construct parking lot and machines’ instructions from a robroute instruction file

As mentioned before, the robroute file includes the width and height of the parking lot and the layout of parking lot. The parking spaces have to be given coordinates in a grid (x and y coordinates). Also, the different types of parking spaces have to be considered as there might be cases where the parking lot is not perfect rectangle without any obstructions in the middle.

The instructions in robroute file are generated by C++ algorithm which uses bitwise addition to characters for differentiating the states. The states need to be converted back from characters to meaningful state enumeration.

Furthermore, the instructions are not explicit as in they do not give an ID to the machine. For visualization purposes they have to be converted to explicit instructions.

### Application is showing static images of start and end state of the simulation

In order to grasp the simulation fully, start and end state of the simulation has to be seen on the screen.

### Application is visualizing the simulation of the algorithm

User has to be able to see all of the steps of algorithm simulation on screen. The simulation has to be visualized continuously – it cannot be step-by-step static images.

### Application is able to scale accordingly to parking lot and viewport size

As the application can be used from variety of devices with different screen resolutions and the parking lot sizes can vary vastly, the application has to be able to scale accordingly.

### User is able to choose the parking lot layout to visualize from application

There are different parking lot simulations that can be visualized. User has to be able to choose between them before the simulation and also change the layout during the visualization.

# System architecture

Technologies used are state-of-art and work for the best result. Web application approach was chosen as in recent years, the browser support for HTML5 technology stack standards has improved drastically, which makes web application developed for HTML5 truly crossbrowser and cross-device experience. The other advantage is the ability to access the application from any location.

Originally, Phaser was chosen as a JavaScript library supporting the visualization part of the development as it has the right support for the requirements of the application. This includes scalability in screen sizes and JSON support. JSON is used as data object transmitting standard as it is quicker to parse and transmit, but also because of its integration in JavaScript language. In the early stages of development process, where the architecture of the application came more apparent, Phaser was ditched from the technology stack as it did not support the core methodologies used in the application.

Python is used as a server side language for its easy to read syntax and possibility to implement new features quickly. Bottle is used as a web framework for it’s lightweightness and simplicity. It has the ability for function-call mapping for clean and dynamic URLs.

## Architecture overview

## Server side platform selection

Nowadays server side programming can be made by many different languages that have different approaches on the web server design patterns. These language include Java, ASP.NET, Python, PHP, Ruby on Rails et cetera. For this project four languages were taken under review that could fill the requirements set by the client side application and the comfortableness of the author in the languages. These languages are Python, PHP, Java and ASP.NET, which builds on C#.

Java is a general-purpose programming languages. The approach in Java is to let developers “write once, run anywhere” [1]. Java is used in a huge variety of use cases, including desktop applications, mobile applications and web applications, client and server side.

PHP is a server-side scripting language which is designed for web development. It can also be used as a general-purpose programming language. PHP is still one of the popular choices of web developers, known for huge community, big variety of frameworks and good documentation. ASP.NET is Microsoft’s open-source web application framework designed to produce dynamic Web pages.

ASP.NET uses C# programming language. A comparative study [2] compares the three aforementioned web technologies using MVC (Model-View-Controller) design pattern on all of the cases to build a web application. In Java that means using JSP web pages (\*.jsp) as views, the controllers are developed using Java servlets and the models are developed using Enterprise Java Beans and Java Persistence API. In ASP.NET, views were developed using Active Server Pages. The controllers are in C# code and views can be developed by either razor or aspx. As PHP does not have its own MVC design pattern, PHP system was implemented using CakePHP framework.

The study concluded that Java processes Login HTTP requests faster than two other systems. The study also shows that Java is the quickest to process GET method requests, but ASP.NET is the quickest in processing POST method requests. All in all, Java and PHP were found to be cheaper to implement compared to ASP.NET. The downside of Java is the difficulty of using it compared to CakePHP and ASP.NET.

Python is a general-purpose dynamic programming language first appeared on 20 February 1991. Amongst others, these are the guiding principles of Python’s design:

* Beatiful is better than ugly
* Explicit is better than implicit
* Simple is better than complex [3]

For web serving purposes, Python uses Python Web Server Gateway Interface (WSGI) as a standard interface between web servers and Python web applications or frameworks [4]. In addition of Python having clean, simple syntax, it also has a wide variety of WSGI frameworks to choose from.

Taking into account the findings in the study, Python is used in the development of server side application for robot car parking system visualization. The reasoning is the advantages over other mentioned programming languages – cleaner syntax without much broilerplate (compared to ASP.NET and Java) and clearer function naming (compared to PHP).

### Python web framework selection

Choosing the right web framework for the application can be troublesome as there are so many different frameworks available. As the application does not need much in terms of web server capabilities itself, the non full-stack frameworks will suit the best. Three of most popular ones are compared: Bottle, CherryPy and Flask.

#### Bottle

Bottle is a lightweight Python micro web framework. It was first released on July 1 2009 by Marcel Hellkamp. It is easy to use, has built-in template engine, Support for JSON client data and can be extended with different plugins. [5] Furthermore, Bottle has a built-in web development server and supports any WSGI capable HTTP server. Built-in web server makes it easy to develop and test the code.

#### CherryPy

CherryPy is an object-oriented web framework for Python. It has a flexible plugin system, powerful configuration system, reliable WSGI thread-pooled webserver amongst other features. CherryPy has been available over ten years and is open-source project. [6]

#### Flask

Flask is a micro framework for Python that is based on Wekzeug and Jinja 2. Flask’s intention is to keep the framework core simple and at the same time extensible, where an extension can be added to the project when it really is needed. This makes Flask minimal and keeps points of failures low. [7]

All of the three aforementioned web frameworks provide similar functionality keeping the application simple without adding any unnecessary overhead. For the application, Bottle was a framework to be chosen. The reasoning was the subjective ease of starting and clearness of documentation by author.

## Data object transmitting standards

The traffic between client and server in the discussed web application in this thesis will, in majority of times, be by asynchronous calls. AJAX [8] standard will be used to accomplish that. There raises a dilemma, which kind of data object transmitting standard to use in this process. The ones under review are the two most popular, XML and JSON.

XML (The Extensible Markup Language) is considered the ‘holy grail’ of computing for its universal data representation format. The priorities when designing the languages were simplicity and human readability. It is primarily used for Remote Procedure Calls. XML does not have any pre-defined tag sets – everything can be configured by user. An example of an object in XML is provided in figure 1.

<person>

<firstname>Suido</firstname>

<lastname>Valli</lastname>

</person>

**Figure 1.** Example of XML object.

JSON is designed to be human readable and easily parsed and used by computers. JSON is directly supported inside JavaScript. JSON is estimated to parse up to hundred times faster than XML. The disadvantages over XML include lack of namespace support, lack of input validation and extensibility drawbacks. An example of an object in JSON is provided in figure 2.

{

“firstname”: “Suido”,

“lastname”: “Valli”

}

**Figure 2.** Example of JSON object.

From the study [9] it can be concluded that sending data in JSON encoding is in general faster than sending it in XML encoding. On the other hand, the transmission times of XML are lower when fever objects are transmitted. As for the fact that sending and parsing JSON data is generally faster than doing the same with XML data and that JSON is natively supported in JavaScript, JSON will be used in the thesis project for transmitting data objects between client application and server.

## Client side platform selection

There are quite a lot web technologies in use to make a graphic animation. Three of the biggest are Flash, Java and combination of HTML5, CSS3 and JavaScript.

Adobe Flash, formerly known as Macromedia Flash and Shockwave Flash, is a platform for creating rich Internet applications, usually used for development of Web-based games and interactive tools. Flash Player plugin is required for Flash content to work as it runs as a client-side sandboxed virtual machine [10]. Downside of Flash is the fact that it is proprietary technology.

Java is a powerful development platform. In Web browsers, Java runs in sandboxed virtual machine. For that fact, Java applets are theoretically platform-independent. The problem with Java is its need for the plugin and the fact that different applications may need different versions of the plugin. Furthermore, the initialization of the applet can be long. Also, there are serious security issues, due to what Java applets are not executed automatically in Web browsers anymore [11].

HTML5, CSS3 and JavaScript (from here on abbreviated to HTML5) provide open source alternative to Java and Flash. HTML5 is used provide static content, CSS3 to style the said content and JavaScript is used to make the content dynamic. As HTML5 is supported by all major browsers without plugin, the users do not have to be worried about security risks. One potential disadvantage of HTML5 would be the different interpretation of the standards by different browsers, but these could be easily overcome with some JavaScript libraries, for example jQuery. The main disadvantage of HTML5 is the relative ease to see the source code. As this thesis is in public domain, this fact is not accounted when choosing the web technologies. As mobile devices have much more interaction channels than traditional network channel, there are new ways for untrusted data to enter mobile devices. When this data is executed, there could be serious consequences. Solutions like PhoneGap are developed to fix the issue. [12]

As the main reason the web application approach for visualization was chosen was the application to run seamlessly on any number of devices, Java and Flash are ruled out mostly because of their plugin requirement, but also for the fact that they are morally old technologies and, as use of them decreases steadily, they might not be supported soon. The technology used for making the visualization application is HTML5 technology stack: HTML5, JavaScript and CSS3

### JavaScript Libraries

JavaScript as a language is very extensive and right now, the language that web programmers use. For visualization, one could start from scratch and invent the logic behind the animation him/herself. This approach is prone to errors and very time exhausting. As there are plenty of different graphics frameworks written on top of JavaScript, it would be wise to study the options and choose the most suitable one.

Ten different options will be studied and best suited for the purpose is chosen. [13]

CAAT (Canvas Advanced Animation Toolkit) is a 2D director based scene graph renderer. CAAT has a chance to have multiple instances as you can have unlimited number of directors for each web page. It is very diverse toolkit with unlimited number of timelines and actors. CAAT has abstracted input system and it does resource management and preloading. Disadvantages of CAAT is that it was last updated over two years ago and it does not have a community behind it.

CakeJS is a scene graph library written in JavaScript meant for HTML5 canvas tags. It has animation timelines and it supports mouse events. It does not support easy resizing of application in different screen sizes and aspect ratios. Furthermore, this project has been archived.

Canvas Engine is a library for easily creating games in HTML5 Canvas. It works on modern browsers and also smartphones. The scenes can be structured and preloaded. In addition, Canvas Engine has a model for handling server-side events to develop a multiplayer game. Although Canvas Engine is meant for building games, it is also suitable for the purpose of robot car parking system visualization and simulation. The main disadvantage is the smallness of community and the fact that Canvas Engine has not been updated for over a year.

ChesterGL (Chester Game Library) focuses on ease of use and performance. It supports a simple scene graph. It has time based actions, different shaders (for WebGL) and batched sprites. The disadvantage of ChesterGL is smallness of community and infrequent updates, lastly updated 2 years ago.

Cocos2d-JS is a game engine for web and native games. It has high performance, has modern JavaScript API, full platform without needing plugins. Furthermore, there is a capability to test and debug the developed application on the browsers before pushing them to target device. It has vast API including transitions, events, persistence et cetera. Cocos2d-JS is a good candidate for the JavaScript library to be used in the thesis practical work.

Construct 2 is designed for creating 2D applications rapidly. It is very high level framework. Construct 2 is not open source nor free as majority of other libraries discussed in this thesis. The major difference of Construct 2 is that it does not require any coding – the primary method of programming is through the event sheets. As Construct 2 is not free and there can be issues interfacing with server that cannot be issued inside the event sheets environment, it is not suitable for the purpose of the application developed in thesis.

Crafty is a flexible 2D framework for JavaScript games. It can either use Canvas or DOM, it has sprite map support and collision detection. It is also lightweight framework and open source. Furthermore, it is actively developed and has active community. All in all, Crafty is a considerable candidate, taking into account the requirements of the application developed.

EaselJS is a 2D graphics engine which supports objects nesting and has mouse interaction model. It uses familiar approach for developers which should make it easy to work with. EaselJS is frequently used in making HTML advertisements. ImpactJS is a 2D and isometric HTML5 graphics engine. Tit supports real time multi user application and implements a scene graph based architecture. It has a built in physics module.

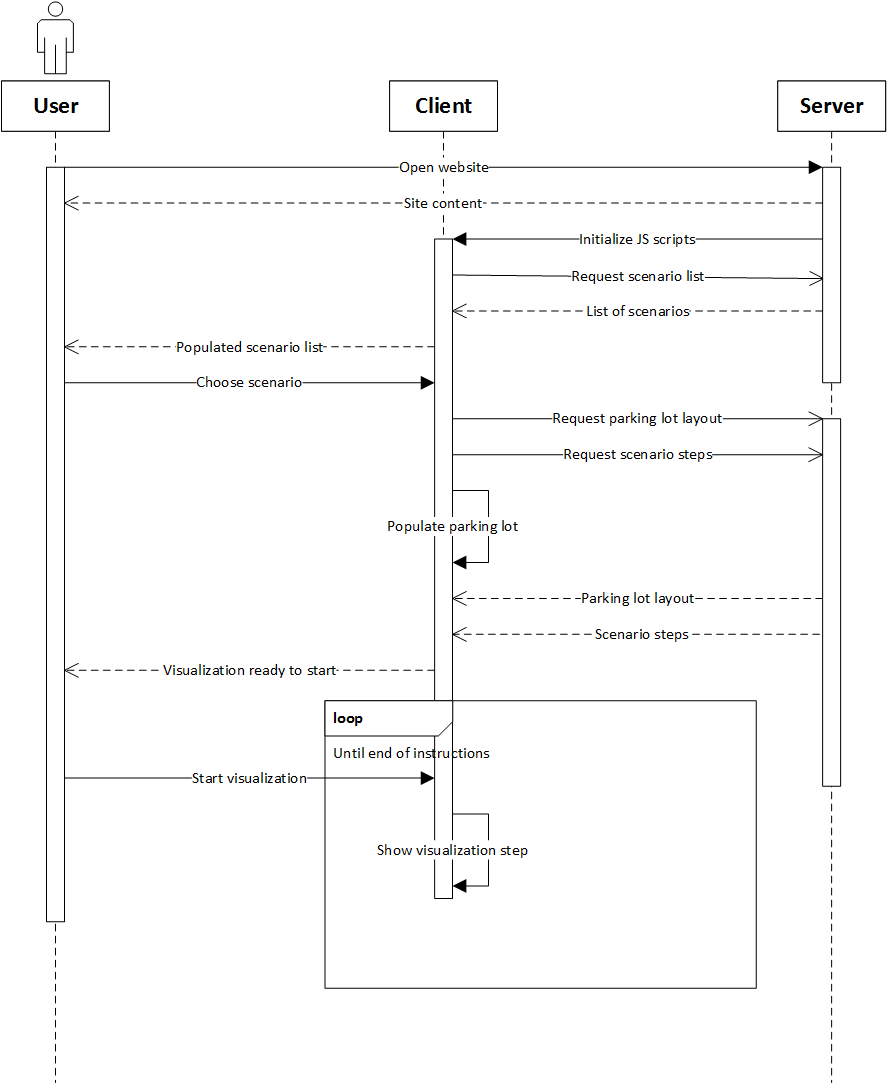
ImpactJS is a considerable framework to use for the application, as it allows creating in addition to two-dimensional scenes isometric scenes, which can be beneficial for understandability of algorithm workings.

Phaser is a free 2D game framework that supports Canvas and WebGL rendering. It uses Pixi.js internally for rendering. Phaser can automatically switch between Canvas and WebGL rendering, according to the available technologies in the device. Phaser can use JSON and XML for asynchronous calls, it supports inputs from mouse, touch screen, keyboard. Furthermore, with Phaser has built-in Scale Manager which allows developer to scale the application to fit any size screen. Phaser is actively developed and has active community.

Although there are plenty of good candidates in the 10 reviewed libraries, the author chose Phaser as it has the most useful features needed for the development of parking system visualization built in. For example, it can use JSON or XML data without any further libraries or development on programmer part.

In the early stages of development it came apparent that the chosen framework is not the best choice for developing an application specified by requirements mentioned in chapter 3.2. The problem is that as Phaser is meant to develop games, there is no way to give instructions where one object has to be on specified frame. After considering alternatives, author ditched the additional layer of 2D framework and used vanilla JS and jQuery to construct a working visualization on HTML5 canvas element.

# Application overview



**Figure 1.** Top level sequence diagram of the application

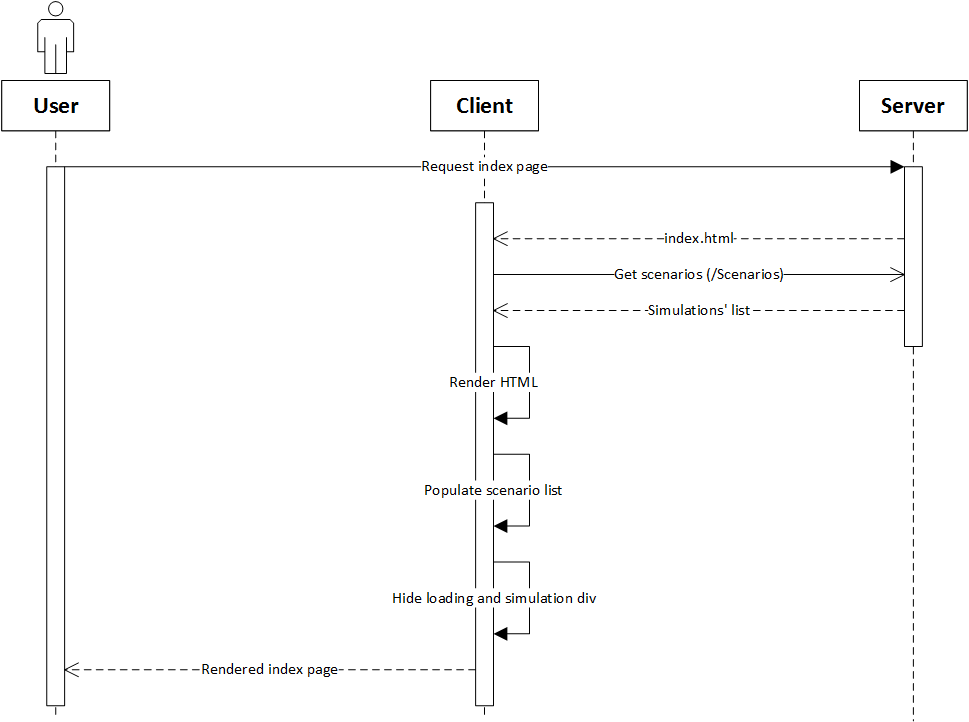
From high level standpoint, the application consists of two separate parts - the client application and server application. Client application is built using HTML5, vanilla JavaScript and jQuery. Server is built using Python 3.x and uses Bottle web framework with added Jinja2 templating library for building HTML page and serving the web content.

As seen from Figure 1 which describes the top level program sequence, after user opens the application website, the static page content is returned and shortly after, the list of scenarios is requested from server by client. When client has chosen the scenario, the layout and instructions are requested from server using AJAX calls to RESTful web service. Once the server has sent the responses for aforementioned calls, visualization page is rendered and ready to visualize the scenario.

In the following subchapters, the detailed flow of the application and application functions are described.

## Application flow

### Rendering the index page



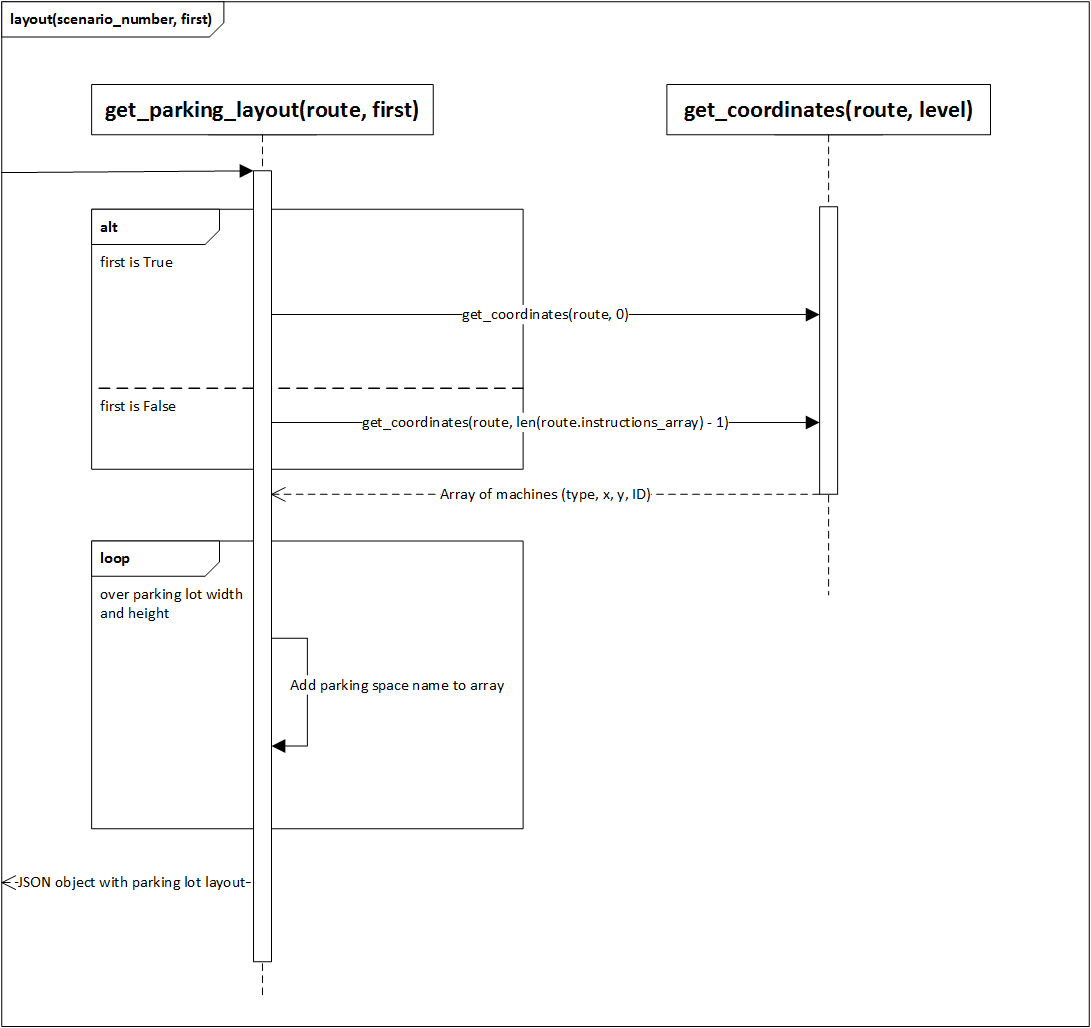
**Figure 2.** Index page rendering sequence diagram.

Application starts by user going to the index page of application web page. For development process, it is <http://localhost:8080>. Upon arrival to the page, the web server will return the index page template. Next, AJAX call is made to /Scenarios page of server to retrieve the available parking scenarios. The scenarios are held in ./Examples folder on server. When server gets a request on /Scenarios address, get\_json\_route\_list() function is called which finds all the files with .robroute extension in Example folder and returns a JSON array with the names and values of the available scenarios.

### Choosing the parking lot and instructions

When the available scenarios have been retrieved, user can choose scenario of what he or she wants to see the visualization of. Also, user has to specify if realistic visualization or visualization with the ability to choose speed is needed. When a scenario gets chosen, the client makes three AJAX calls to server. Firstly, to /Scenarios/<scenarioNr>/Layout/0 to get the end state of the visualization and shortly after that /Scenarios<scenarioNr>/Layout/1 to get the starting state. The need for two calls is to show the user static images of the parking lot at the beginning and end of visualization at all times. Third AJAX call is made to /Scenarios/scenarioNr/Instructions/Realistic to get the instruction steps for the parking lot. Instead of keyword Realistic 0 or 1 is used – when user requested realistic visualization, the keyword is 1, otherwise it is 0.

#### Retrieving parking lot layout.



**Figure 3.** Sequence diagram of server side coordinates retrieval

Server, upon getting a request to return parking lot layout, calls get\_parking\_layout(route) function. Route is an object that is returned by function get\_route(filename).

get\_route(filename) reads the file with the name given in function argument and returns Route class object. Route class object consists of three parts:

* lot\_size, which is an array with two values – number of parking spaces in parking lot vertically and horizontally,
* lot\_layout, which is a two-dimensional array that holds the parking space’s state (to which directions is it able to move from this parking space),
* instructions\_array, which is an array of instruction steps that hold the four different states for each parking space in each array.

get\_parking\_layout(route) returns a JSON object describing the width and height of the parking lot concerning the parking spaces, layout with parking spaces states and machine array that describes the position of the machines in the grid, their ID and their sprite type. Detailed diagram of the function can be seen on figure 3.

Machine array is retrieved from get\_coordinates(route, level, making\_instructions = False) function which iterates over the scenario’s instructions at the given level. When it finds a machine on any parking space, it will add its type (either car or robot), coordinates, unique ID and sprite image type to the array that is going to be returned in the end of the iteration.

Example of server’s response for layout request:

{

"layout": [

["wallNW", "wallN", "wallN", "wallNE"],

["wallW", "nowall", "nowall", "wallE"],

["wallW", "nowall", "nowall", "wallE"],

["wallSW", "wallS", "wallS", "wallES"]

],

"machines": [

["C0", 0, 1, "C0"],

["R", 1, 2, "R0"]

],

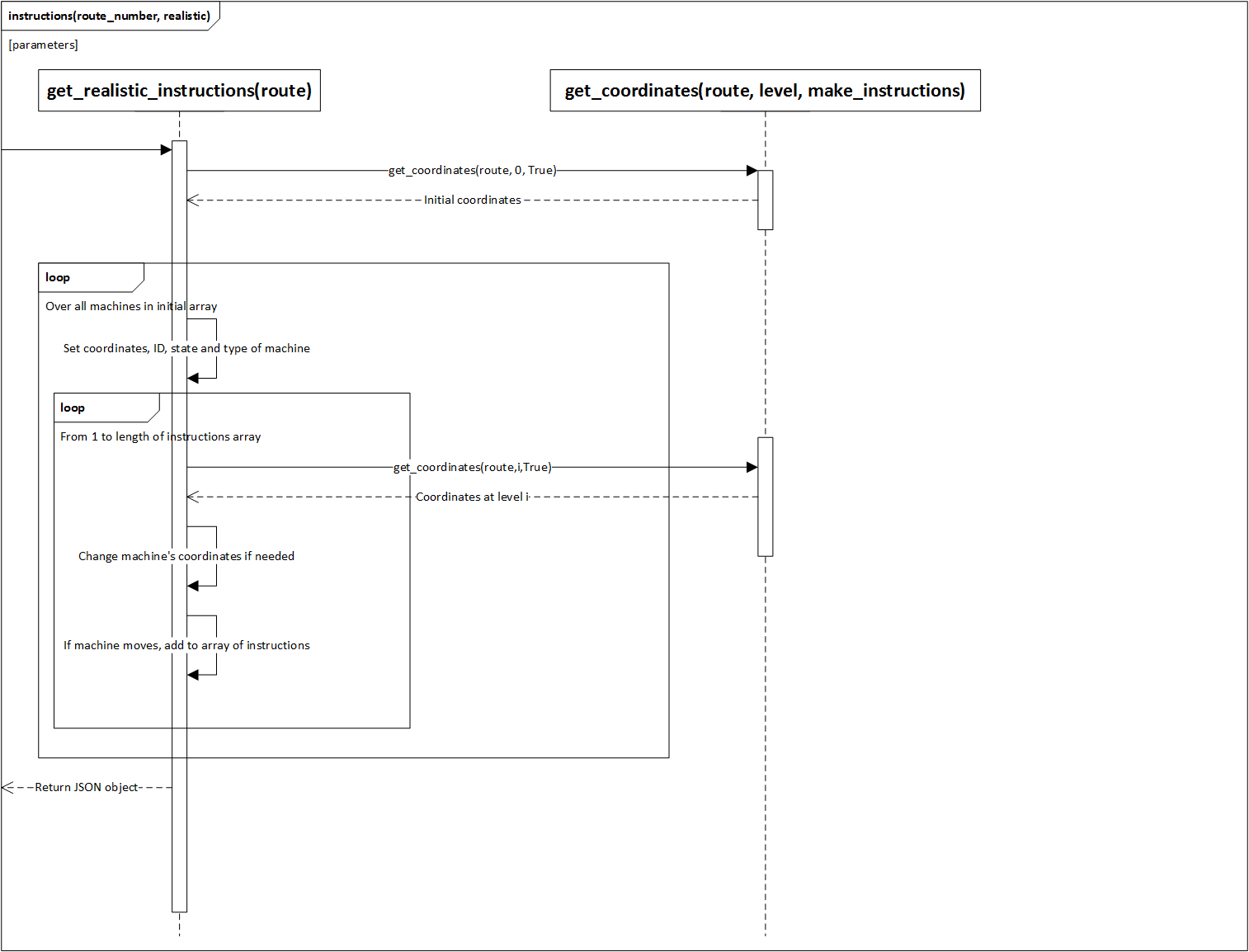
"width": 10,

"height": 4

}

**Code example 1.** Example of layout JSON response

#### Retrieving instructions



**Figure 4.** Sequence diagram of server side instructions retrieval

Function get\_instructions(route) is called upon server getting a request to /Route/routeNr/Instructions. The function will take initial machine array as a starting point by calling function get\_coordinates(route ,0, True). Adding True as third argument will add the four states to every machine’s array. Next, the function loops over every machine in the machine array and goes over all the instruction steps. When looping over the instructions, it keeps record of machine’s ID and coordinates and will change the machine’s coordinates accordingly to the states. If machine is moving at some level, the machine ID, instruction direction and speed is added to the instruction array.

The function returns the array of explicit instructions needed by the client to move the machines accordingly as a JSON object. The example of said JSON object can be seen at code example 2.

[

[

["R0", "W", 2],

["R1", "S", 4]

],

[

["R0", "W", 1],

["R1", "S", 2]

],

[

["R0", "W", 1],

["R1", "S", 4]

],

[

["R0", "W", 1]

],

[

["R0", "W", 1]

]

]

**Code example 2.** Example of instructions JSON response.

### Rendering the main workspace

After client application receives the instructions and parking layout JSON objects, it does the following

1. Renders the final state of scenario on canvas, makes it an image and adds it to the menu,
2. Renders the start state of scenario on canvas, makes it an image and adds it to the menu,
3. Hides loading div, shows visualization workspace
4. Starts visualization loop

The application only requests the types of parking lot images from the server that are needed for the scenario. Retrieved images are cached in an array to keep network traffic low. Furthermore, the parking space is rendered in two or three layers, depending on if the user wants to see the grid on the parking lot or not. This solution allows to change the underlying asphalt or concrete pattern with minimal effort – only one image needs to be changed.

### Visualizing the scenario

After the rendering of the scenario, user has two options: either to start automatic visualization or step-by-step visualization. Automatic visualization can be changed to step-by-step visualization and vice versa during the scenario. The visualization works on frame basis. That means that each frame simulationLoop() function is called. Said function controls all of the visualization. Firstly, it increments variable STEPS by one if there is any movement on the screen. After that, it will clear the canvas element from everything and calculates the instruction array index from the STEPS variable. If the function discovers that it is time for new instructions, it will firstly stop all the movement on the screen by calling stopAllMovement() function. Next it will call moves(routeInstructions, step) function that gives new instruction to all the machines which will move at next instruction step. After that, parking lot is rendered by calling createParkingLayout() function, then robots are rendered calling renderMachines(robots) and lastly, cars with renderMachines(cars). Last but not least, frames per second are calculated in the simulationLoop() function. As simulationLoop() calls window.requestAnimationFrame(simulationLoop) function as the first thing, the next time that browser is ready to get the next frame, simulationLoop() is called again.

#### Function moves()

Function moves(instructions, stepNr) takes the array of instructions and the step of instructions as arguments. Function checks if the new step is in the bounds of instructions array. If it is, it will iterate over all of the instructions in that step and calls moveMachine(machine, instruction, speed) function which gives the machine object direction on what to do next. This function is described in details in the next chapter. moveMachine(…) function also changes the text shown to the user describing what machines are moving where at what speed in that step.

#### Function moveMachine()

Function moveMachine(machine, instruction, speed) gives instructions to the machine object. It goes through the different instruction cases and updates the x and y coordinate direction of the machine, the speed and calls the machine’s update function. The sprite class, that all machines have, will be covered in detail in next subheading.

#### Class sprite()

Class sprite(options) is the class for all machines. It holds all the settings of the machine, most important of them are the width, height, image, x coordinate, y coordinate and moving, lifting flags of the machine. The class also has methods render(), update(dirX, dirY) and addPixel(dirX, dirY, updateshadow = true). Method render() renders the machine itself and its shadow. Method update will increment the machine’s speedCount variable and will call addPixel() method when the machine’s direction, speed and speedCount variables match the set conditions are matching. For north, east, west and south movement the machine will be moved certain amount of pixels away from its current location. When machine is lifting or dropping, the machine is, in addition to coordinates movement, scaled up or down, depending on the vertical movement.

## Application functions

In this subchapter, the functions of the application are described. The functionality is divided into two parts – the index page functionality and visualization page functionality.

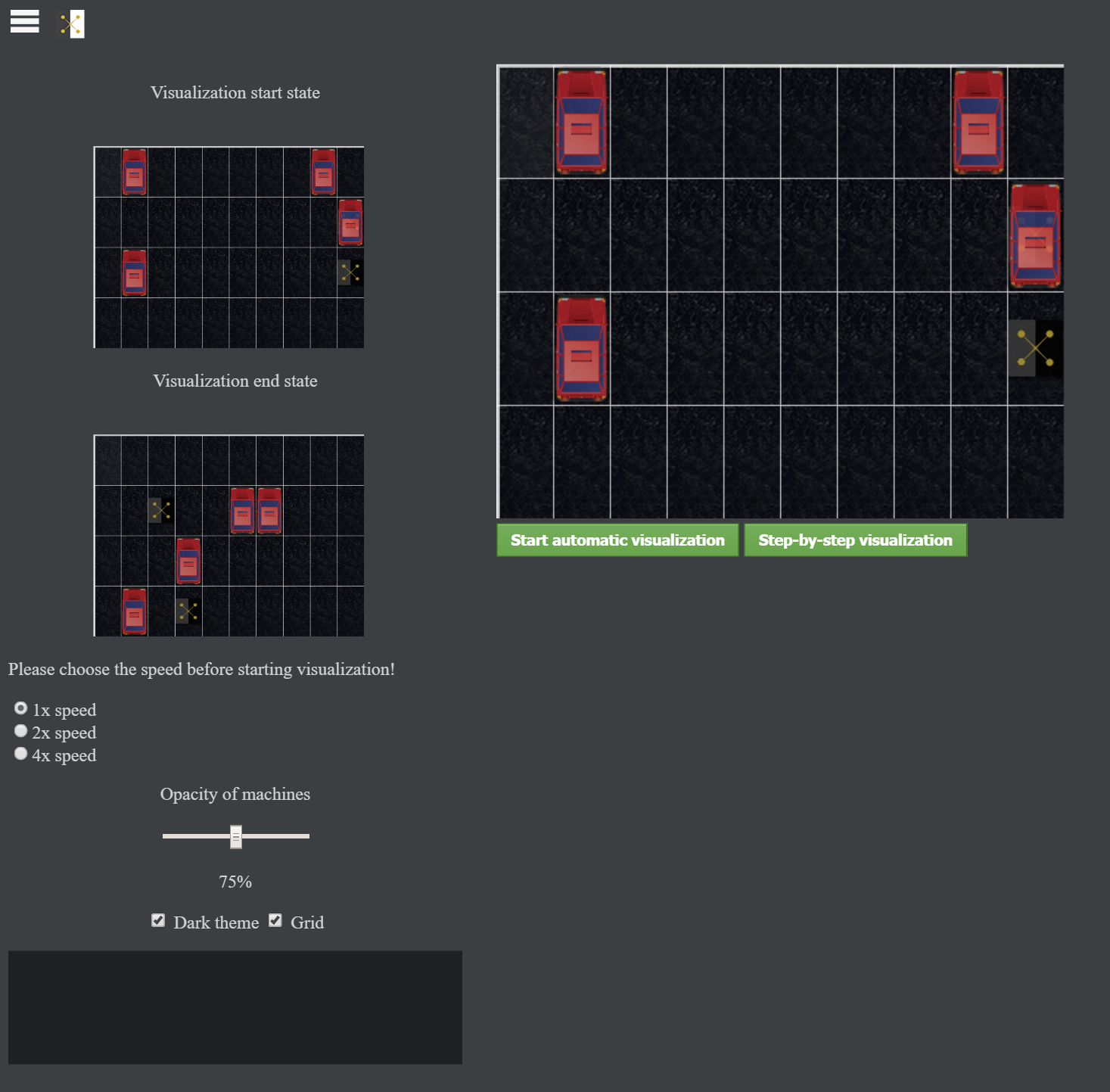
### Index page functionality



**Screenshot 1.** Index page of the application

On the index page, user can choose between different visualization scenarios that are present in the server. User also has the opportunity to choose if the visualization is going to be realistic or there is option to change the speed of the visualization. Realistic mode means that the machines are accelerating and decelerating as cars and robots do in real life. In the mode where it is possible to choose the speed, machines move in a set speed.

### Visualization page functionality



**Screenshot 2.** Visualization page with settings menu expanded.

On the visualization page, user has chance to change several settings of the visualization. Settings can be seen by clicking on the hamburger button in the top left corner of the page. By clicking on the robot button on its right, user is taken back to the index page where he or she can choose new scenario. Firstly, the opacity of the machines can be changed from 50% to 100% by moving the slider to left or right. The current opacity is shown under the slider. Default opacity is 75%. The purpose for this functionality is for the user to see the robots under the cars, if he or she feels the neccessity for that.

The theme of the page can be changed from dark to light by toggling the theme checkbox. Default value is dark theme. This changes the background of the page, the parking lot image and the text color according to the theme settings. New themes can be easily made for the program doing minimal changes in the source code.

Furthermore, users have option to turn the grid on the parking lot on or off. By default, grid is turned on as it makes following the visualization easier.

When the user had selected the option to choose the speeds of the simulation, there is option to choose if the visualization is going to be made at original speed or two or four times the original speed. Once the visualization has started, it is not possible to choose the speed anymore.

Also, visualization start and end states are shown on the side of the vizaulisation canvas for better understanding what the algorithm has to do.

When visualization is running, the information about the movement on the current step is shown. The information shown is the ID of the machine, it’s movement direction (north, west, east, south, lifting or dropping) and the speed or level of the machine. The speed of the machine shows how many steps it is needed for the machine to get from one space to the adjacent space. In case of vertical movement such as lifting and dropping, it shows the level of progress the machine is at. For example, „Machine R0 makes step L 1“ means that Robot with ID „R0“ is lifting and is on level 1.

Once the visualization has started, the frames per second are shown. The purpose of this is to indicate wether the machine running the visualization is up to the task – if the FPS drops under 24, user experience can suffer. The lower limits of the machine are determined in validation chapter.

To start automatic visualization, user has two options – either to click on the „Start automatic visualization“ button or to click on any car on the parking lot. On user input, the visualization starts. The automatic visualization can be changed to step-by-step visualization by clicking on relevant button at any time during the visualization.

Starting step-by-step visualization is done by the „Step-by-step“ visualization button. The button will be disabled, colored red and the text on the button is changed to „Active movement“ until the step is finished. After the step is finished, it turns back to active button with text „Proceed to next step“. Now user can watch the next step whenever it is convenient for him or her.

# Validation

In this chapter, the application’s working on different platforms, its efficiency and integrity are validated. As the application consists of client and server parts, then two of them are tested separately in all of the categories.

## Visual testing

The overall workings of the application are tested by visual observation – if everything looks and application functions work as specified.

Platforms on which server side of application was tested are as follows: Windows 10, Linux Mint 17.3 “Rosa” and Ubuntu 14.04.

Windows 10 was running on a physical PC with following specifications:

* Intel core i5-3570K @ 3.4 GHz
* 12 GB DDR3 RAM
* nVidia GeForce GTX 660 Ti
* 120 GB Samsung SSD 840 series

Linux Mint 17.3 “Rosa” was installed on a virtual machine with 4 GB of RAM and 128 MB 3D hardware graphics acceleration.

Ubuntu 14.04 is running as a virtual machine in Nitrous.io platform, which is a development environment in the cloud.

Client side web browsers used were as follows:

* Linux Mint
  + Firefox 42.0
* Windows 10
  + Google Chrome 50.0.2661.102 m
  + Firefox 46.0.1
  + Microsoft Edge 25.10586.0.0
* Mac OS X 10.11.4 (CPU – Intel core i7 @ 2.2 GHz, 16 GB RAM)
  + Safari
* Android (OnePlus 2, Android version 6.0.1)
  + Google Chrome 50.0.2661.89
* iOS (iPhone 4S, iOS 9.2)
  + Safari

By combining the server platforms with client platforms and web browsers, following test combinations were tried:

|  |  |  |  |
| --- | --- | --- | --- |
| Test scenario number | Server platform | Client operating system | Browser |
| 1 | Linux Mint | Linux Mint | Firefox |
| 2 | Windows 10 | Windows 10 | Google Chrome |
| 3 | Windows 10 | Windows 10 | Firefox |
| 4 | Windows 10 | Windows 10 | Microsoft Edge |
| 5 | Ubuntu | Linux Mint | Firefox |
| 6 | Ubuntu | Windows 10 | Chrome |
| 7 | Ubuntu | Android | Chrome |
| 8 | Ubuntu | iOS | Safari |
| 9 | Ubuntu | OS X | Safari |

**Table 1.** Test scenarios for application

During the testing of scenarios, all of the functionality was tested and the behaviour of the application asserted. Furthermore, the average FPS was captured.

All the tests were successful and while the appearance of some HTML elements changed depending on the browser, everything worked as specified. In the beginning of testing, it came apparent that Safari and Microsoft Edge are not supporting default parameters in function arguments as described in ES6/ES2015[[4]](#footnote-4). Mentioned browsers raised an exception when parsing following code:

that.addPixel = function (dirX, dirY, updateShadow = true) {

…

}

**Code sample 3.** Parameter default value

To mitigate the problem, the functions with default parameter values had to be changed as follows:

that.addPixel = function (dirX, dirY, updateShadow) {

updateShadow = typeof updateShadow !== 'undefined' ? updateShadow : true;

…

}

**Code sample 4.** Alternative to parameter default value.

The average FPS of the test scenarios can be found in table 2. The FPS in modern browsers is usually capped at 60 frames per second.

|  |  |
| --- | --- |
| Test scenario number | Average FPS |
| 1 | 58 |
| 2 | 60 |
| 3 | 60 |
| 4 | 60 |
| 5 | 58 |
| 6 | 60 |
| 7 | 58 |
| 8 | 32 |
| 9 | 60 |

**Table 2.** Average FPS on visualization in test scenarios.

As can be seen from table 2, almost all the browsers were able to run the visualization with 60 FPS, where browsers limit the framerate. Only test case where FPS was lower by significant margin, was test case number 8 where iPhone 4S was used as a test device. Considering that the device was released in 2011, has dual-core 1 GHz processor and 512 MB RAM, this result was expected. In addition, 32 average FPS does not make user experience any worse.

## Efficiency of application

The efficiency of client application is partly covered in previous chapter showing the average framerate of the visualization. The application’s network efficiency and CPU load are tested in this chapter. For server, the average time for serving JSON is tested.

### Client side efficiency

For client side efficiency, Google Chrome developer tools are used to check the CPU load and network efficiency.

Network efficiency is measured by recording network log with Chrome developer tools. The recording is started when the user goes to the index page of application and ended after full visualization. In addition, the theme and grid settings are toggled during the time to ensure maximum network load. The results are as follows: 34 requests were made and 362 KB of data was transferred from server to the client. The content was loaded in 1.46 seconds. Those values are in limits with the requirements from chapter 3.

Measuring the load to the processor uses the same test scenario as network efficiency test. The test output has been taken from Google Chrome task manager. Application’s tab uses 34,468 MB of memory and when the visualization is ongoing, it takes 2-4 % of the CPU time. Considering that most contemporary computers have at least 4 GB of RAM and smartphones 2 GB, the program is efficient in using the device’s resources.

Server side efficiency is measured by finding the average time to get scenario list, layout and instructions as a response to requests. The percentage of CPU utilization by the server process is also captured. All of the test scenarios made 100 requests to the server. The test scenarios and results can be found in table 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test scenario number | Request | Average time in milliseconds | Responses under 250/300/350 ms | CPU utilization % of the process |
| 1 | Get list of scenarios | 151 ms | 77/97/100 | 0.2 % |
| 2 | Get ending state layout for first scenario | 178 ms | 62/86/91 | 9.8 % |
| 3 | Get starting state layout for first scenario | 177 ms | 67/90/93 | 10.5 % |
| 4 | Get realistic movement instructions for first scenario | 236 ms | 66/89/93 | 23.7 % |
| 5 | Get normal movement instructions for first scenario | 235 ms | 61/88/92 | 23.1 % |

**Table 3.** Server efficiency test results.

As seen from the test results, getting the list of scenarios took on average 151 milliseconds and 97% of requests were done under 250 milliseconds. Server used 0.2% of CPU time for this request on average.

There is no statistical difference in requesting the starting and ending state layouts of the scenarios, with the average response time being 177 milliseconds, 86% of responses being under 250 milliseconds. The server was using roughly 10% of CPU time dealing with the requests.

Requesting the movement instructions took on average 235 milliseconds to respond with 88% of responses being under 250 milliseconds. Application utilized on average 24% of CPU while processing the requests.

Finally, all of the test scenarios were run sequentially 1000 times. The average response time was 231 milliseconds and the CPU utilization 11%.

## Integrity

The integrity of the application is tested by using unit tests on server and visual confirmation on client side.

The unit tests on server side cover functions get\_parking\_layout(), get\_instructions(), get\_realistic\_instructions() and get\_json\_route\_list(). Unit tests are situated in tests.py file.

# Future opportunities

# Conclusions

# References

|  |  |
| --- | --- |
| [1] | N. Langley, „Write once, run anywhere?,“ May 2002. [Võrgumaterjal]. Available: http://www.computerweekly.com/feature/Write-once-run-anywhere. [Kasutatud 14 04 2016]. |
| [2] | D. A. Botwe ja J. G. Davis, „A Comparative Study of Web Development Technologies Using Open Source and Proprietary Software,“ *International Journal of Computer Science and Mobile Computing,* kd. IV, nr 2, pp. 154-165, 2015. |
| [3] | T. Peters, „PEP 20 -- The Zen of Python,“ 19 August 2004. [Võrgumaterjal]. Available: https://www.python.org/dev/peps/pep-0020/. [Kasutatud 20 April 2016]. |
| [4] | P. J. Eby, „PEP 333 -- Python Web Server Gateway Interface v1.0,“ 07 December 2003. [Võrgumaterjal]. Available: https://www.python.org/dev/peps/pep-0333/. [Kasutatud 20 April 2016]. |
| [5] | M. Hellkamp, „Bottle: Python Web Framework,“ 19 April 2016. [Võrgumaterjal]. Available: http://bottlepy.org/docs/dev/index.html#. [Kasutatud 20 April 2016]. |
| [6] | The CherryPy team, „CherryPy -- A Minimalist Python Web Framework,“ 2015. [Võrgumaterjal]. Available: http://www.cherrypy.org/. [Kasutatud 20 April 2016]. |
| [7] | A. Roacher, „Foreword -- Flask Documentation (0.10),“ 2013. [Võrgumaterjal]. Available: http://flask.pocoo.org/docs/0.10/foreword/#what-does-micro-mean. [Kasutatud 20 April 2016]. |
| [8] | J. J. Garrett, *Ajax: A New Approach to Web Applications,* San Fransisco: Adaptive Path, 2005. |
| [9] | N. Nurseitov, M. Paulson, R. Reynolds ja C. Izurieta, *Comparison of JSON and XML Data Interchange Formats: A Case Study,* Montana: Montana State University, 2009. |
| [10] | N. Stewart ja S. Reimers, „Presentation and response timing accuracy in Adobe Flash and HTML5/JavaScript Web experiments,“ *Behavior Research Methods,* pp. 309-327, 2015. |
| [11] | N. Ganesh, *Static Analysis of Malicious Java Applets,* San Jose: San Jose State University, 2015. |
| [12] | P. Trivedy ja S. Harneja, „HTML5 – the new standard for Interactive Web,“ *International Journal of Research,* kd. 1, nr 10, pp. 39-42, 2014. |
| [13] | AIR FORCE RESEARCH LAB ROME NY INFORMATION DIRECTORATE, *Advanced Visualization and Interactive Display Rapid Innovation and Discovery Evaluation Research Program task 8: Survey of WEBGL Graphics Engines,* New York: AIR FORCE RESEARCH LAB ROME NY INFORMATION DIRECTORATE, 2014. |

Appendix

1. Glossary

|  |  |
| --- | --- |
| Caret  The bar (or other symbol) marking the active editing point. | Sisestusmärk  Märk, mis märgib teksti sisestamise asukohta. |
| Template  A gauge, pattern, or mold, commonly a thin plate or board, used as a guide to the form of the work to be executed. | Mall  Näidik, muster või valuvorm, mis esitab täitmisele võetava töö struktuuri. |

1. License

**Non-exclusive licence to reproduce thesis and make thesis public**

I, **Suido Valli**,

(*author’s name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

**Visualization of simulations of a robot operated car park system**,

*(title of thesis)*

supervised by Dirk Oliver Theis,

*(supervisor’s name)*

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **14.05.2016**

1. <https://github.com/qiao/PathFinding.js> [↑](#footnote-ref-1)
2. <https://qiao.github.io/PathFinding.js/visual/> [↑](#footnote-ref-2)
3. <http://visualgo.net/> [↑](#footnote-ref-3)
4. <http://wiki.ecmascript.org/doku.php?id=harmony:parameter_default_values> ECMAScript parameter default values [↑](#footnote-ref-4)