TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SAMPO LAHTINEN

UTILIZATION OF GAME ENGINE IN SIMULATION VISUALIZATION

Master of Science thesis

Examiner: prof. Hannu Koivisto
Examiner and topic approved by the Faculty Council of the Faculty of Engineering Sciences on 4th May 2016

ABSTRACT

SAMPO LAHTINEN: Utilization of Game Engine in Simulation Visualization
Tampere University of Technology
Master of Science Thesis, 78 pages, 6 Appendix pages
November 2016
Master's Degree Programme in Automation Engineering
Major: Information Systems in Automation
Examiner: Professor Hannu Koivisto

Keywords: game engine, visualization, simulation, container terminal

Modern game engines provide software developers with comprehensive toolsets for turning their visions into visually appealing 3D worlds. In addition to gaming industry, these frameworks can be used in creating visualizations for real-world processes. In this thesis, the concept of game engine based industrial process visualization is demonstrated in the context of automated container terminals. During the thesis project, a real-time 3D visualization tool was developed that can be used to visualize simulated terminals and actual systems. The work was commissioned by a Finnish client company, working in the cargo handling industry.

The thesis document comprises of a background part and a solution part. In the background part, the most important concepts of container terminal operations are presented. The focus is then moved to the software systems that are used in the automated terminals provided by the client company. The background part contains also a review of the existing 3D applications in the container handling industry and a literature survey of various other projects, which are utilizing game engines for simulation purposes.

For the practical part, a requirements analysis was performed for the visualization tool. The development platform was then chosen by comparing two of the most commonly used modern game engines: Unity and Unreal Engine. While both of the engines had their advantages and disadvantages, Unity was chosen as the development platform for several reasons: It allowed using the existing 3D models of the client company without doing any manual conversions to the files. The object model and scripting system of Unity was also regarded as intuitive and easy to use. Finally, the software framework used in Unity allowed easy integration with the software systems of the client company.

The implemented application is configured by using similar XML files that are used in GUI applications of actual terminals. It communicates with the terminal automation system by using the common communication platform. Machine positions and container events are acquired real-time from the automation system. It was also proven, that the application can be extended to send messages back to the automation system.

The solution was tested with a virtual container terminal, including 10,000 containers and 47 container handling machines. It was confirmed, that the application is able to handle large amount of concurrent movement without problems. However, the vast amount of objects in the terminal makes the visualization of the whole area a challenging task for a conventional PC. Further graphical optimization is required in order to provide sufficient frame rate and smooth animation in all situations.

TIIVISTELMÄ

SAMPO LAHTINEN: Pelimoottorin hyödyntäminen simuloinnin visualisoinnissa
Tampereen teknillinen yliopisto
Diplomityö, 78 sivua, 6 liitesivua
Marraskuu 2016
Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Automaation tietotekniikka
Tarkastaja: Professori Hannu Koivisto

Avainsanat: pelimoottori, visualisointi, simulointi, konttisatama

Nykyaikaiset pelimoottorit tarjoavat ohjelmistokehittäjille kattavat työkalut visuaalisesti näyttävien 3D-ympäristöjen luomiseen. Pelinkehityksen lisäksi näitä työkaluja voidaan käyttää tosielämän prosessien visualisointiin. Tässä diplomityössä pelimoottorin käyttöä teollisten prosessien visualisoinnissa on demonstroitu automaattisilla konttiterminaaleilla. Työssä toteutettiin reaaliaikainen 3D-visualisointityökalu, jota voidaan käyttää simuloitujen järjestelmien ja oikeiden terminaalien visualisointiin. Sovellus tehtiin tilaustyönä suomalaiselle lastinkäsittelyalalla toimivalle yritykselle.

Työ jakautuu taustaosioon ja toteutusosioon. Taustaosiossa esitellään konttiterminaalien toimintaa yleisesti, minkä jälkeen tutustutaan asiakasyrityksen automaatiojärjestelmissä käytettäviin ohjelmistoihin. Taustaosio sisältää lisäksi kirjallisuusselvityksen olemassaolevista 3D-työkaluista kontinkäsittelyalalla sekä yleisemmin projekteista, joissa pelimoottoria on hyödynnetty simulointitarkoituksiin.

Käytännön osiossa tehtiin vaatimusmäärittely visualisointityökalulle, minkä jälkeen toteutusympäristö valittiin vertailemalla kahta yleisesti käytettyä nykyaikaista pelimoottoria: Unitya ja Unreal Engineä. Molempien ympäristöjen hyvistä ja huonoista puolista huolimatta Unity osoittautui selkeästi paremmaksi valinnaksi projektin toteuttamiseen: Se mahdollistaa asiakasyrityksessä tehtyjen 3D mallien hyödyntämisen sellaisenaan ilman manuaalisesti tehtäviä tiedostomuunnoksia. Unityn käyttämä oliomalli ja ohjelmointirajapinta koettiin vaivattomiksi käyttää verrattuna Unreal Engineen. Lisäksi Unityn käyttämä ohjelmistoalusta mahdollisti helpon integraation asiakasyrityksen järjestelmiin.

Toteutettu sovellus käyttää konfigurointiin samanlaisia XML tiedostoja, joita käytetään todellisten konttiterminaalien valvomosovelluksissa. Se kommunikoi automaatiojärjestelmän kanssa käyttäen yrityksen yhteistä kommunikaatiorajapintaa. Koneiden paikkatiedot ja konttitapahtumat saadaan reaaliaikaisesti automaatiojärjestelmältä. Sovellusta voidaan myös tarvittaessa laajentaa lähettämään viestejä automaatiojärjestelmän suuntaan.

Ratkaisua testattiin virtuaalisen konttiterminaalin kanssa, joka sisälsi 10 000 konttia ja 47 kontinkäsittelykonetta. Testillä osoitettiin, että sovellus kykenee käsittelemään ongelmitta useiden koneiden samanaikaista liikettä. Koko terminaalialueen visualisointi osoittautui kuitenkin haasteelliseksi tehtäväksi tavalliselle PC-tietokoneelle, koska piirrettävien kohteiden määrä terminaalissa on erittäin suuri. Sovelluksen graafista toimintaa tulee optimoida, jotta voidaan varmistaa riittävä kuvanopeus ja sulava animaatio kaikissa tilanteissa.

PREFACE

This thesis was commissioned by the Kalmar branch of Cargotec Finland Oy. I would like to express my gratitude to Mr. Hannu Santahuhta from Kalmar for offering me the topic. This project serves as a fascinating example of how software can be used to create elegant solutions to real-life problems. The thesis process involved acquiring and applying knowledge of maritime container logistics, software system integration and 3D graphics, among many other things. This project contributed greatly to my pursuit of creating software not only for the sake of software itself but as a part of larger value creation process.

I would also like to thank Engineering Manager Petteri Kylliäinen for supervising the thesis project from the Kalmar side and Professor Hannu Koivisto from TUT for the academic guidance and examination of the thesis. Developing of the application and producing of a high-quality thesis document proved to be a surprisingly time-consuming task. The whole process from receiving the topic to publishing the final thesis version took almost ten months. I would like to thank all parties for their patience. I also want to thank the whole software development team of Kalmar for their invaluable help in understanding of how the terminal software works.

The previous year has brought some of the biggest changes in my life, since I started my studies at TUT in 2009: Returning to Finland after a year-long exchange in Germany, seeing the old student comrades start new lives in other cities, moving from the student role to a new challenging position in working life, and writing the final thesis. All these events have been characterized with a feeling of letting go of the old and stepping into a new era, filled with excitement but also with a great deal of uncertainty.

I want to thank my beloved girlfriend and my family for supporting me all the way through this process. Special thanks go also to my friends in the student choir Teekkarikuoro for providing the much-needed recreation and a feeling of continuity during these turbulent times.

Tampere, 9th of November 2016

Sampo Lahtinen

CONTENTS

NOTATIONS AND ABBREVIATIONS

Boldface	Term definition
Italic	Reference to a program, software object, data field or file
[ ]	Literature reference
( )	Cross-reference or explanation
`Courier New`	Program code or XML data
`//`	Begins comment line in program code
`<!-- -->`	Marks comment in XML data
Cambria Math	Mathematical variable or equation

AGV	Automated Guided Vehicle
AI	Artificial Intelligence
ALV	Automated Lifting Vehicle
ASC	Automatic Stacking Crane
ASCCS	Control System for Automatic Stacking Cranes
AutoStrad	Automated straddle carrier concept from Kalmar
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
CAVE	Cave Automatic Virtual Environment
CHE	Container Handling Equipment
CPU	Central Processing Unit
CS	Control System
DLL	Dynamically Linked Library
EIS	External Interface Service
EMS	Equipment Monitoring System
FPS	Frames Per Second
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HT	Horizontal Transportation
HTCS	Horizontal Transportation Control System
IDE	Integrated Development Environment
LOD	Level Of Detail
MVC	Model-View-Controller, a software design pattern
.NET	Software framework developed by Microsoft
PLC	Programmable Logic Controller
Qt	A software framework, used in creating GUI applications for various platforms
RAM	Random-Access Memory
RMG	Rail Mounted Gantry crane
RTG	Rubber-Tired Gantry crane
SC	Straddle Carrier
STS	Ship-To-Shore crane
TCP/IP	Transfer Control Protocol / Internet Protocol, the standard technology suite for implementing communication in computer networks

TEU	Twenty-foot Equivalent Unit
TLS	Terminal Logistic System
TOS	Terminal Operating System
UE	Unreal Engine
UI	User Interface
UniQ	Communication framework used in Kalmar products
VPN	Virtual Private Network
VR	Virtual Reality
XML	Extensible Markup Language

1. INTRODUCTION

Modern game engines provide software developers with comprehensive and powerful toolsets for converting their visions into visually appealing 3D worlds. In addition to gaming industry, these frameworks can be used in creating visualizations for real-world processes. 3D visualizations play an increasingly important role in the marketing of industrial products, and the ability to provide a convincing visualization may even become one of the key factors in winning or losing a customer project. Visualizations are also used commonly together with simulations to provide better understanding of the system that is being simulated. In today's networked automation environments, real-time data from the process can be used as data source for the visualization.

In this thesis, the concept of game engine based industrial process visualization is demonstrated in the context of automated container terminals. Volume of the worldwide container traffic has been growing steadily since the introduction of standard shipping containers in the 1960s. While the container ships have become increasingly larger, efficiency of harbour systems has become a key success factor for container terminal operators. Software and automation play a prominent role in a modern harbour, where unmanned machines are performing concurrent moves side-by-side with the manual operations. Integration of various subsystems calls for lots of configuration work from the terminal automation providers. Significant advantages can be achieved by using simulation models, which may be used in planning of new terminals and optimizing of logistics systems in existing facilities. When the simulation models are being combined with actual control software, entirely functional virtual terminals may be created prior to construction of any physical terminal equipment.

Goal of the thesis project was to develop a prototype for a 3D visualization tool, which can be used to visualize virtual container terminals. These virtual terminals consist of simulated process models and actual control system software, which is used to control the logistics process. They communicate with the visualization application by using the same communication platform that is used in actual terminals. The featured container handling machines and software systems are developed by Kalmar, which is a branch of Cargotec Corporation. Cargotec is a Finnish company, providing intelligent cargo handling solutions for a wide range of industries.

The rest of this thesis can be roughly divided into a background part and a solution part. The background part comprises of chapters 2–4. In Chapter 2, a general introduction to container terminals is given. The logistics process of a terminal is explained and the most important machine types and concepts are presented. A brief overview is also giv-

en of the various planning problems that may be addressed by using simulation models. In Chapter 3 the focus is moved to software systems used in Kalmar solutions. The presented software components and communication concepts will provide the basis for the data interchange between the visualization application and the virtual terminal process. Chapter 4 provides an overview of existing 3D applications, including the 3D visualization tools used by Kalmar and the virtual terminal solutions provided by other companies. The chapter contains also a literature review of the previously done simulation projects utilizing game engines in fields outside container handling industry.

The solution part of the thesis covers the specification, design, implementation and testing of the visualization tool. Requirements analysis for the application is performed in Chapter 5. In addition to the functional requirements, certain technical aspects are discussed and a preliminary architecture is presented for the application. In Chapter 6 the concept of game engine is covered in more detail and a comparison is made between two of the most commonly used modern game engines: Unity and Unreal Engine. The implemented solution is presented in Chapter 7, which describes the structure of the application, handling of the terminal objects in 3D world and the animation system. In Chapter 8, testing procedures are described and further development ideas are given for the visualization tool. Final conclusions of the thesis are presented in Chapter 9.

2. MARITIME CONTAINER TERMINALS

The invention and standardization of shipping containers has had a tremendous impact on the cargo handling industry for the last half of a century. The first regular container shipping service started in 1961 between the US East coast and ports in the Caribbean, Central and South America. By year 2000 over 60% of all deep sea cargo in the world was transported in containers, with containerization rate up to 100% on some routes. [1] Today the standardized steel boxes are ubiquitous in sea, rail and road transportation of cargo. Storage capacities and traffic volumes are commonly expressed in twenty-foot equivalent units (TEU), where one TEU represents a 20 feet long container. Other commonly used container lengths include 40 and 45 feet, which both measure as 2 TEU [2]. The volume of global container traffic has been growing steadily also during the 21st century: Between years 2000 and 2014 the maritime container traffic in the world tripled from around 225 million to 680 million TEU per year. [3; 4]

Along with traffic volumes, the sizes of container ships have been growing dramatically from several hundred TEU of the early generations to over 18.000 TEU of the largest vessels of today [5]. Operating costs of a ship grow steeply with its size, but the ship only generates revenue when it is moving [5; 6]. Thus, having short turnaround times in harbours has become an increasingly important success factor for shipping lines. Competition between container terminal operators has created demand for more and more efficient port systems, allowing short waiting times and fast loading and unloading of vessels, while keeping the operating costs moderate. [2] Computer systems and automation have become key elements in both equipment control and high level management of a terminal. Today's automated harbour is a complex environment where sophisticated software systems are used for logistics planning, communication and safe and reliable execution of concurrent movements.

Generally the term container terminal may refer to any facility where containers are being moved from one transport method to another, e.g. from truck to train. Some maritime terminals may also act as pure interchange terminals, connecting major ocean lines to regional shipping routes or inland barge transports. In this thesis the focus is on terminals which have both access to road or rail network and services for marine vessels. Section 2.1 describes the basic operations of such terminal. Commonly used container handling equipment (CHE) types and terminal automation issues are discussed in section 2.2. The related logistics planning problems and software systems are presented briefly in section 2.3.

2.1 Terminal operations

Container terminals vary a lot in terms of size, layout, equipment and degree of automation. While it is not possible to provide one general blueprint for a terminal, each terminal can be roughly divided to certain functional areas, as illustrated in Figure 1.

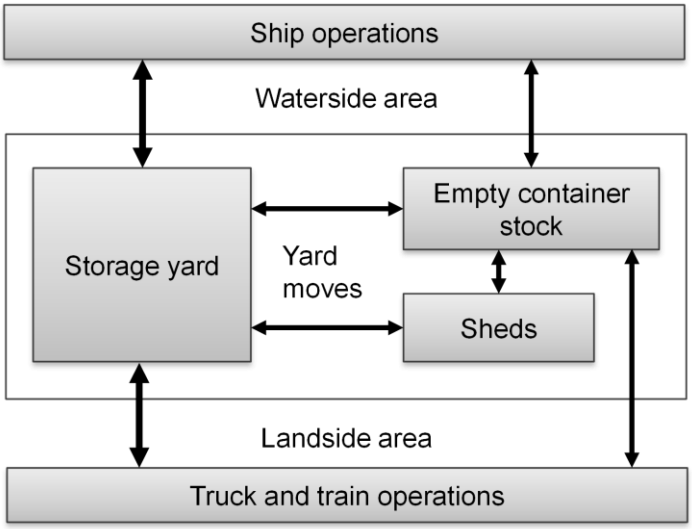

Figure 1: *Functional areas of a maritime container terminal. Adapted from [2].*

Loading and unloading of vessels takes place next to the quay wall, in an area called waterside or quayside. In modern terminals, the container movements between vessel and apron are performed by a rail-mounted crane, commonly known as ship-to-shore crane (STS). Typical STS has a fixed orientation towards the ship and it is driven sideways on the rails along the quay wall. The crane uses a moving trolley and hoist to pick up the containers from the ship and ground them on the apron, and vice-versa. In terminals with low container throughput, STS cranes may be replaced by traditional quay cranes or on-board lifting equipment of vessels [6].

Containers are transported to and from waterside area by means of horizontal transportation (HT) equipment. For HT equipment there are several options, like shuttle carriers, trailers pulled by terminal tractors, or automated guided vehicles (AGV). Some of these are able to pick and ground containers by themselves, while trailers and AGVs are dependent on a crane to do the loading and unloading for them.

Storage yard is used for stocking incoming containers before they are transported onwards. Containers are typically organized in stacks, which may have different heights and shapes. Stack positions are commonly expressed in terms of row, bay and tier, representing the logical position of the container in three dimensions. In addition to stacks, the yard may have a separate area for storing empty containers and sheds for temporary storing of goods.

Serving of trucks and trains takes place in the landside area, also known as hinterland area. Separate equipment may be required for transporting containers between the storage yard and landside, and for loading and unloading of vehicles.

One possible operational scheme for a container terminal is presented in Figure 2. In the given terminal, rail mounted gantry cranes (RMG) are used for stacking containers in the yard area. HT equipment is utilized in both ends of the chain and separate gantry cranes are used for serving trucks and trains.

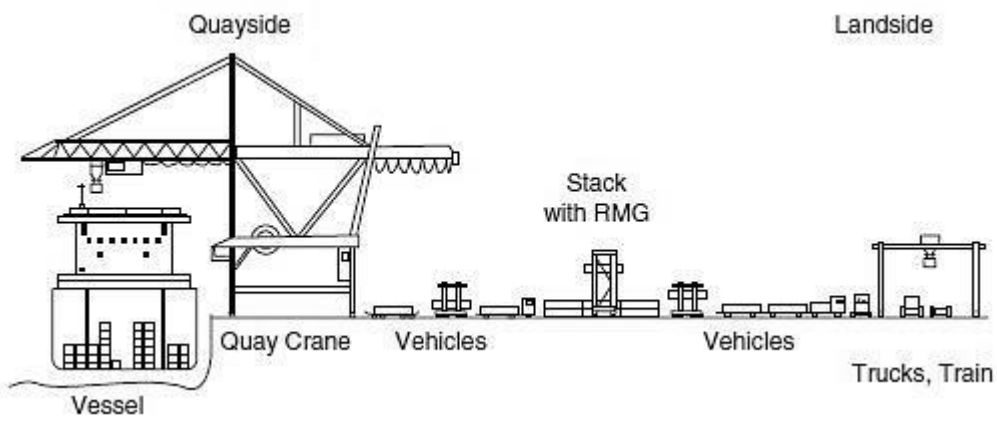

Figure 2: *Operational scheme example for a container terminal* [2]

2.2 Equipment and automation

The most commonly used CHE types include ship-to-shore cranes, rail mounted and rubber-tired gantry cranes, shuttle and straddle carriers, terminal tractors, reach stackers and masted container handlers. [7]. Many of these are available as fully or partially automated versions. An automated rail mounted gantry crane is commonly known as automatic stacking crane (ASC). Unmanned versions of the straddle and shuttle carriers are sometimes referred to as automated lifting vehicles (ALV), whereas a driverless chassis without lifting capabilities is called automated guided vehicle (AGV).

The choice of equipment for a terminal depends on several factors, including planned terminal capacity, planned container throughput and possible space restrictions. Term stacking density is used to measure the utilization of land area, commonly expressed as TEU per hectare. In sparsely built terminals, containers may be stored on chasses instead of stacking them on top of each other [2]. While this system can work on relatively inexpensive equipment, it requires lots of space, resulting in low stacking density. A medium-density solution is achieved by using straddle carriers or reach stackers, which are able to stack the containers up to 3–4 tiers high. In medium-sized to large terminals, straddle carriers may be used as the only CHE type besides the STS, as is demonstrated in Figure 3. In this system, containers are stored in linear stacks, with gaps between rows to allow straddle carriers to reach the inner containers of the stack. [6]

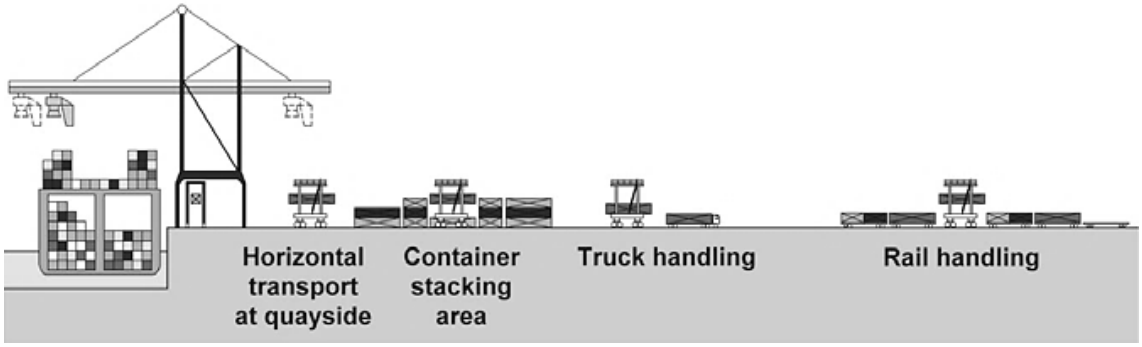

Figure 3: *Example of "pure SC" system, utilizing straddle carriers for stacking, horizontal transportation and landside operations [6]*

The highest stacking density and container throughput is achieved with gantry cranes. Gantry cranes stack the containers in tightly organized blocks, consisting of 7–8 tiers at most. Rubber-tired gantry cranes (RTG) are able to navigate between multiple blocks, whereas an RMG can operate in a single block with up to 12 rows of containers. A stacking system based on RMGs is relatively expensive to deploy and rigid in terms of terminal layout changes. On the other hand, it provides the best land area utilization and highest productivity for large and very large terminals. [6]

The appropriate degree of automation for a terminal depends e.g. on the planned container throughput and labor costs of the region. According to estimates by Kalmar, in highly developed western countries the cost of labor can account to more than 50% of the total operating costs [8]. Manual operations are also prone to human errors, resulting in accidents, mistakes, and losing of containers. A fully automated system – consisting of unmanned CHE – has high initial costs due to special requirements for the terminal infrastructure and the required control systems. Thus, it is often not feasible choice for small or medium-sized terminals. Partially automated systems can be used to increase productivity in manual terminals, e.g. by assisting CHE drivers in their work and providing the operators in the control room with comprehensive data about the situation in the terminal.

Later in this thesis, a control software framework and visualization model is presented for automatic stacking cranes and straddle carriers. Their physical features and working principles are described in the following sections.

2.2.1 Automatic stacking cranes

Automatic stacking crane (ASC) is a Kalmar brand name for fully automated rail mounted gantry crane systems. Typical ASC block is situated perpendicular to the quay wall, with one end next to the landside and other next to the waterside. There are usually two or three cranes per block (see Figure 4). In two-crane configuration the both cranes operate on same rails and cannot pass each other. The three-crane configuration has one

larger crane operating on a separate set of rails. This larger crane is able to pass both smaller cranes, allowing more concurrent movements in the block. The three-crane system is however more costly due to the extra crane and the required rail installations. [8]

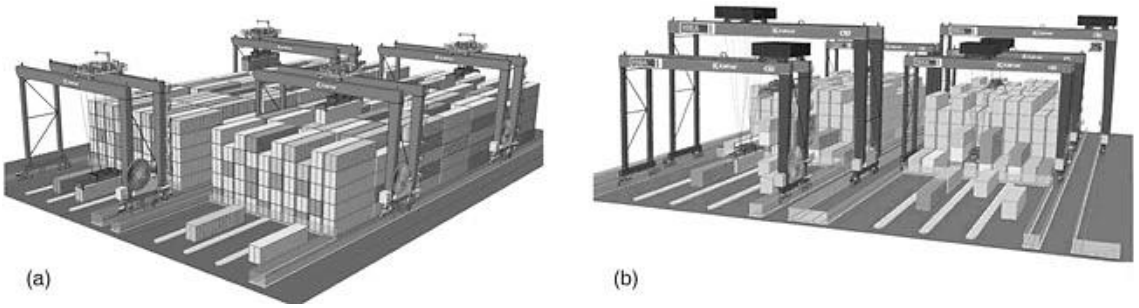

Figure 4: *a) 3D model of two ASC blocks with two cranes per stack. b) Ditto with three cranes per stack. [8]*

Both ends of an ASC block have interchange areas for other equipment. The landside interchange area is typically organized in truck lanes, where road trucks are served directly by an ASC crane. Incoming trucks are identified by using light gates and automatic measurement systems. Picking and grounding of containers is performed semi-automatically by using so called truck driver kiosks for passing orders to the crane.

The waterside interchange area has lanes for HT equipment. ASCs are usually combined with shuttle or straddle carriers, which are able to pick and ground containers independently. Thus the ASC crane may leave the container in the interchange area, where it is later picked by another CHE, or vice versa. During quiet times the ASC cranes are used to move and sort the existing containers in the stack to enable shortest possible performance times for the upcoming tasks.

An ASC crane consists of a gantry, a trolley, a hoist and a spreader. The gantry represents the rigid frame structure of the crane, which moves linearly on the rails. The trolley moves perpendicular to the gantry, allowing transitions between rows. Hoist is used for moving the spreader up and down. Thus, the positions of gantry, trolley and hoist represent the three degrees of freedom in the ASC block.

The spreader is used for lifting the containers. Typical telescopic spreader consists of a rigid main body and two retractable end components, which are moved to make the spreader length match the container. Figure 5 shows a close-up view of a spreader.

Containers are attached to the spreader by using twistlocks. Each corner of a container has a fitting, which allows a twistlock to be inserted in a certain position. The twistlocks are then turned to lock the container securely to the spreader. Similar twistlock systems are used in ships, trains and truck trailers to hold the containers in place during transportation.

Figure 5: *Close-up view of a spreader, with ASC cranes in background [9]*

Spreaders are used in nearly all CHE types that are able to do independent lifting. Some spreaders are able to pick two 20 feet containers simultaneously, which is known as twin-lifting. These spreaders have an extra set of twistlocks in the middle to grasp the inner corners of the two containers. Some terminals use double spreaders, which allow side-by-side lifting of two 40 – 45 feet containers, known as tandem lifting [5]. Furthermore, some RMG systems have rotating spreaders, which allow picking of containers in different angles and loading them on the train with doors facing each other [10]. These special spreader types are however not common in ASC cranes.

2.2.2 Straddle and shuttle carriers

Straddle carrier (SC, or a straddle) is a freely moving CHE type, which has a wheel span slightly wider than a single container. Its hoist is fixed to its main structure and cannot be moved sideways like in gantry cranes. Thus, an SC is only able to drive over individual rows of containers and handle them from above. Typical SC has a maximum stacking capability of 1-over-3, meaning that it is able to drive above 3 tiers of containers while carrying one [6].

Shuttle carrier (shuttle) is basically a 1-over-1 SC, designed for fast horizontal transportation. While SCs can be used for stacking (as illustrated earlier in Figure 3) shuttles may only operate in single-tier rows, where all containers lie on the ground level. Typical ASC terminal by Kalmar uses shuttles for transporting containers between the ASC

block and the working area of the corresponding STS. Kalmar shuttle carriers are also capable of twin lifting [11].

In a paper from J. Pirhonen, a comparison is made between an ASC and shuttle based container handling system and a more traditional concept, which is using RTGs and terminal tractors. The simulated shuttle system uses equipment pooling, where any of the automated shuttles can be assigned to any task, when it becomes available. According to Pirhonen, the shuttle concept requires less than 50% of the transport equipment when comparing to traditional systems utilizing AGVs or terminal tractors. As the latter are not capable of independent lifting, they may spend over 50% of their total working cycle waiting for a crane to perform a lift or ground operation for them. The system based on ASCs and automated shuttles is also told to produce less emissions and require far less operating staff than a system based on manual equipment. [8]

Figure 6 below shows an automatic straddle carrier at TraPac terminal in Los Angeles, USA. The featured terminal uses both ASCs and automatic straddle carriers for stacking, and the straddles also act as horizontal transportation vehicles, serving both stack types [12]. A container handling system consisting of ASC cranes and automated shuttles or straddles is used as an example case throughout the rest of this thesis. The software framework for controlling these machines is presented later in Chapter 3.

Figure 6: *Automatic straddle carrier at TraPac terminal in Los Angeles, USA [9]*

2.3 Logistics planning

Managing a modern container terminal is a very complicated task due to the strict performance requirements and the large amount of concurrent operations to be handled. Long-term planning of operations requires cooperation and data exchange between terminal operators, shipping lines and landside transport companies. Position of each container should be known to the operators from the moment it enters the terminal area until the point it leaves in a ship, train or truck. Furthermore, the equipment in the terminal should be utilized in such manner, that waiting times and logistics related costs are minimized for all parties.

Operational planning in a container terminal can be divided to several processes:

- **Berth allocation**: For each arriving vessel, a quay position needs to be allocated so that the ship can be served as quickly as possible. The berthing schedule is normally planned weeks in advance, but the system should be able to tolerate sudden changes, e.g. due to ship delays. In an optimal solution, the transport distances for incoming and outgoing containers are minimized for all vessels. [2; 13]
- **Stowage planning**: Stowage plan defines the positions of containers in a ship. The initial planning is done by the shipping company, while deciding the schedule for a voyage. Container slots in the vessel are assigned according to containers' properties and their destination ports. The resulting initial plan is sent to terminal operators, who use it to create the loading and unloading plan for the actual containers. Ultimate goal of stowage planning is to maximize the ship's utilization and minimize extra container moves in the port while maintaining the stability of the ship. [2; 13]
- **Crane allocation**: For each vessel, a certain number of STS cranes are assigned and the planned container moves are scheduled to them. The chosen cranes must be able to reach the given quay position and work with the given ship, taking into account the size of the crane. The choice of cranes for one ship affects not only that vessel but the overall traffic situation in the terminal, as the chosen cranes cannot be used to serve other vessels. [2; 13]
- **Storage and stacking logistics**: This process covers the reservation of yard capacity and choice of storage positions for the containers. In the ideal case, the containers should be positioned so that the need for repositioning or stack reshuffling is minimized. The chosen storage slots should also lie relatively close to the berthing position of the corresponding ship. In practice, the containers often arrive at the terminal lacking accurate data about the onward transportation method or other details. Due to the amount of uncertainty in the process, a good stacking strategy and forecasting methods are required for successful operation. [2; 13]

- **Transport optimization**: The task of transport optimization can be divided to optimization of horizontal transportation and optimization of stacking operations. The former can be divided further to landside and waterside transportation, if HT equipment is utilized in both areas. General goals of the optimization include maximizing CHE productivity and minimizing waiting times for all parties. Increasing the number of CHE may have a negative effect to the terminal efficiency, as it increases the congestion in the terminal. Intelligent job allocation and CHE routing are often more efficient and economical methods. [2]
- **Landside operations planning**: Landside operations concern the service of trains and road trucks. The planning tasks include the allocation of rail tracks and truck positions, and assignment of the container moves to the related CHE. While the planning problems are similar to those in ship operations, the landside operations are usually less critical, as containers are not stacked on vehicles, and waiting delays are usually more acceptable. The planning horizon is also shorter than in seaside operations: Truck operations are often not planned until the truck arrives at the landside area. [13]
- **Workforce planning**: Workforce planning covers the determination of overall workforce capacity for the terminal and the scheduling of individual labor tasks to employees [13]. Even in fully automated terminals, skilled operators are required for keeping the operations running. STS cranes and supporting CHE types are typically operated by drivers. Some special containers types are also handled manually by employees. These include the refrigerated "reefer" containers, which are stored in designated racks and connected to electric supply.

When looking at the range of planning problems, along with the questions considering terminal layout and choice of equipment, the need for simulation technologies becomes obvious. Furthermore, the vast amount of process data makes computerized control system a necessity for a terminal to succeed. In large terminals, the number of container movements per day exceeds 10.000 [2]. Thus, the planning systems must accommodate to the constantly changing situation in the terminal.

In automated terminals, the logistics planning and administrative functions are bundled in a business system known as Terminal Operating System (TOS). Typical functions of a TOS include handling of the discussed planning processes, reporting, invoicing, document management, gate control, messaging, etc. An advanced TOS can be configured to schedule container moves automatically according to its optimization logic and the defined business rules. Some of the modern terminal operating systems include N4 by Navis, Master Terminal by Jade Software, and TOPS by RBS. [14-16]

While TOS is the system mainly responsible for the logistics process and financial functions of a terminal, separate automation systems are typically required for the monitoring of equipment and execution of operations ordered by TOS. The integrated solution by Kalmar is presented in the following chapter.

3. KALMAR AUTOMATION SOFTWARE PLATFORM

The terminal automation solution provided by Kalmar is known as Terminal Logistic System (TLS). TLS is a distributed solution, which in practice consists of several software systems, performing various tasks. An example of TLS software stack, tailored for ASCs and automatic straddle carriers, is presented in Figure 7.

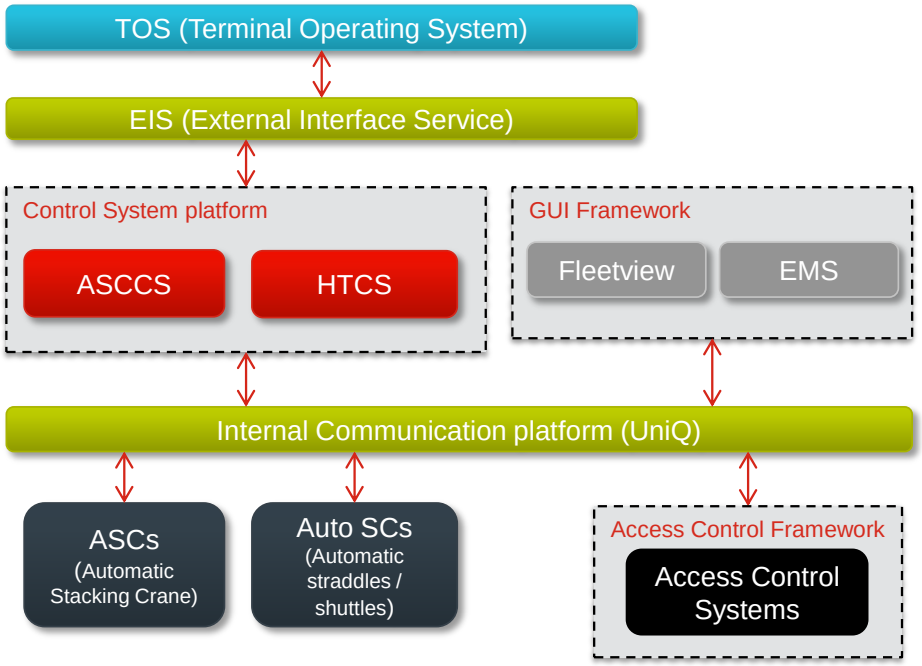

Figure 7: *Kalmar TLS software stack for automatic stacking cranes and straddle carriers. Adapted from [17].*

TLS is designed to be combinable with any TOS. Decoupling of TLS components from third party applications is performed in External Interface Service (EIS) platform. Terminal specific integration modules are encapsulated in the EIS layer, and other TLS applications only depend on the services of EIS. Thus, the internal data structures of TLS can be kept unchanged from project to project.

Execution of automated operations results from cooperation between the on-board automation systems of the CHE and the various control systems running on server computers. The most important parts concerning the 3D visualization tool are the control systems built on Control System platform, the internal communication framework known as UniQ, and the existing UI applications implemented on GUI framework. The rest of this chapter is organized in three sections, describing each of these concepts.

3.1 Control System platform

Control System (CS) platform provides the software components, that are used to control a single machine or (more typically) a group for machines. It can be thought as an executive layer between the TOS and the machine-level automation systems of CHE. Typical ASC terminal has multiple ASC Control Systems (ASCCS) installed, while the horizontal transportation is organized under one Horizontal Transportation Control system (HTCS), which controls all HT equipment in the terminal.

A CS is responsible for converting the work orders from the TOS into specific instructions for the CHE under its control. One of its main tasks is to prevent collisions and deadlock situations resulting from multiple CHE trying to access the same area [10]. Before a machine is allowed to move, its CS makes a routing decision and space reservation in cooperation with other control systems. Space reservations are used to ensure that each machine has the necessary space to perform its operation, and no-one else has the same space. Separate access control systems are used to secure areas, where humans or manual equipment may enter the automated zone. Such areas include service gates, truck lanes, and interchange areas, where containers are transferred between automatic and manual equipment. [17]

Each CS has an internal world model, which describes the containers, stack positions, obstacles and different zones in its operational area. Some of this information is specified in configuration files, but the situation may also change dynamically e.g. due to maintenance work performed in certain areas of the terminal. [17] Information of the handled containers is stored in the database of the corresponding CS. This causes special situations in interchange areas, which belong to the operational area of two or more control systems: A container lying in the interchange area of two systems has a data representation in both systems.

Control systems monitor the status of their respective CHE and store it in a dedicated data structure on the server [18]. This server-side data can be used for fetching and visualizing the CHE status instead of connecting to the physical machine. This is especially useful in simulated systems, where the physical machines are replaced with simulation models. These so called simulation stubs are included in ASCCS and HTCS to allow easy building of test scenarios in virtualized server environments.

3.2 Internal communication platform

The internal communication platform – known as UniQ – acts as a cross-platform data distribution layer, which provides a common way of communication between TLS applications. UniQ framework implements a standardized and highly reliable messaging service, which is based on common data semantics. [17] It is also a highly modular and

customizable system, where new services can be added without changing the configuration of the existing system [19].

UniQ networks consist of independent service instances called peers. Each peer is able to act as publisher or consumer of data. Peers communicate by creating a communication channel, which is a virtual two-way connection between the two peers. After the channel is created, peers can send messages to other peers or order messages from the other peers. In order to create a channel, the creating peer has to know the name of the remote peer. [19]

The data that is exchanged in the UniQ framework is organized under a common naming system, known as tagging. The term tag stands for the type of a message and tagged item refers to the actual data content of the message. Each tag has a name, description, and further metadata associated to it. This information cannot however be sent along with the tagged items due to network bandwidth restrictions. To overcome this problem, all the known tags are enumerated and only the corresponding identification numbers are delivered with the tagged items. Tag metadata is synchronized globally in a configuration file known as tag map. [20] Program 1 presents an example of one tag in *tagmap.xml*, describing position information for a crane.

```
<?xml version="1.0" encoding="UTF-8"?>
<tagmap>
    <tag id="315035">
        <name>CRANE.POSITION.X</name>
        <desc>Local X-position of the crane</desc>
        <quantity>length</quantity>
        <unit>mm</unit>
        <struct>uint</struct>
    </tag>
</tagmap>
```

Program 1: *Example of tagmap.xml, containing information for one tag*

In addition to the tags, UniQ framework defines alarms and events, which can be sent from different actions. Delivery of alarms and events involves no data content, that is, only the necessary identification information is sent [20]. Descriptions of known alarms and events are defined in *alarmlist.xml* and *eventlist.xml* respectively [19].

UniQ messaging is implemented by a cross-platform software component known as tag facade. Tag facade is written with Qt and implemented as a dynamically linked library (DLL). Its responsibilities include the validation and interpretation of messages according to the configuration files, establishing of communication channels between peers and handling of the communication through the lower layers of the UniQ communication stack. [20]

The data distribution protocol of UniQ is based on group communication system provided by the Spread Toolkit. Spread Toolkit is an open source software package,

providing reliable and scalable messaging services for distributed network applications [21]. Spread implements a daemon-client model, where message distribution, ordering and group memberships are handled by server processes known as daemons. Client applications connect to the daemons, which act as message brokers between clients. Establishing of a UniQ channel requires a daemon that is accessible to both peers. Thus, the network location of the daemon needs to be known, when setting up UniQ communication. [17]

3.3 GUI framework

The GUI framework consists of platform independent user interface (UI) components and a common back-end that is used to establish UniQ communication for the GUI applications. The GUI components are implemented by using Qt framework and C++ as the primary programming language. Due to the cross-platform nature of Qt, the same graphical widgets may be used to build monitoring applications for the control room operators and on-board touchscreen solutions for the CHE drivers.

An application for monitoring the state of a single CHE unit is called Equipment Monitoring System (EMS). EMS applications are commonly installed in on-board computers of the respective CHE and they use UniQ to communicate with the automation systems of the vehicle. EMS can also be installed on control room computers to allow remote monitoring of CHE.

A control room application named Fleetview combines EMS functions with a comprehensive 2D view of the terminal area. Figure 8 shows an example of a terminal overview in Fleetview. A machine view of an individual ASC crane is shown in Figure 9. In addition to these views, Fleetview offers several operational views, which describe the traffic situation in the terminal in more detail. The views are updated according to the tagged data from the UniQ platform and the movements of the machines are animated at a rate resembling real-time motion. Besides monitoring, the application can be used for sending commands and creating work orders for equipment, allowing semi-manual operations for automated terminals.

Fleetview implements effectively in 2D many of the features that are expected from the 3D visualization tool. Thus, it is used as a reference solution for the specification, design and testing of the 3D application later in this thesis. In the following sections, a closer look is taken into some important aspects of the GUI implementation.

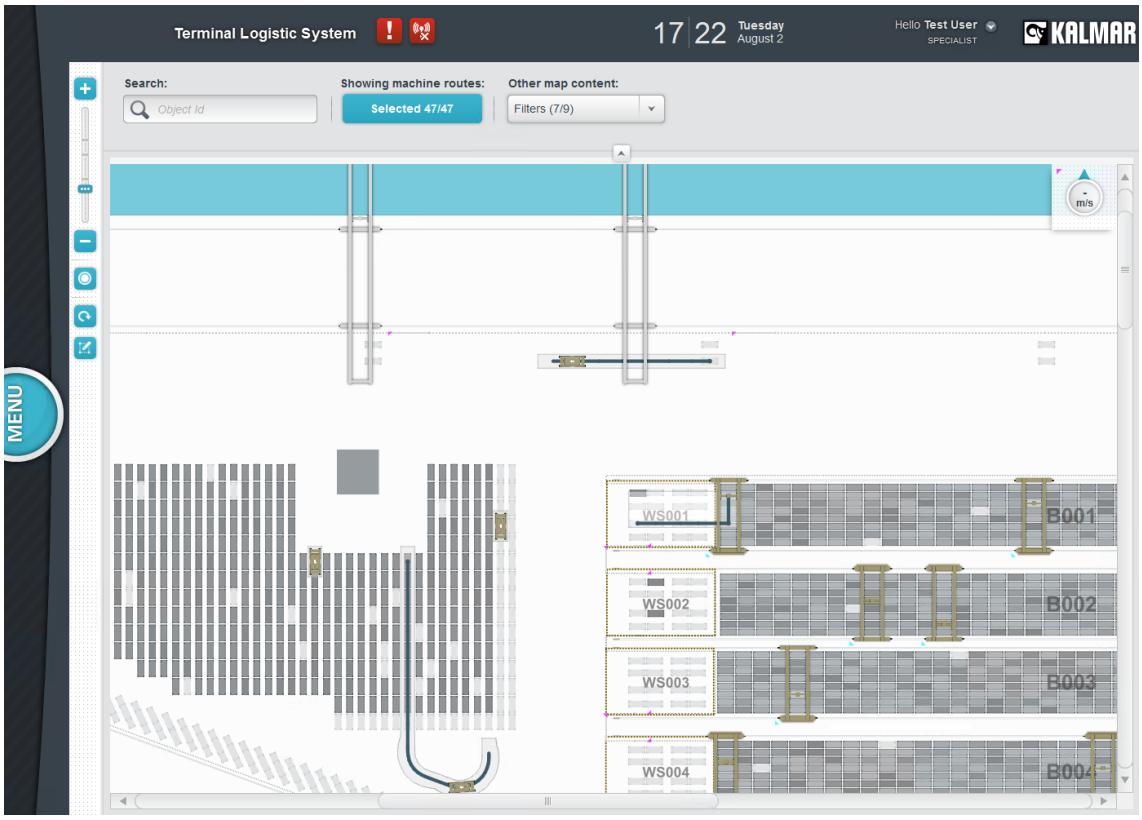

Figure 8: *Terminal overview from Fleetview*

Figure 9: *Machine view of an ASC crane in Fleetview*

3.3.1 Coordinate mapping

In order to visualize objects in their correct positions, the GUI needs a common localization system that can be applied to all objects in the terminal. Automated vehicles are commonly equipped with GPS systems and different kinds of sensors to keep track of the machine's location and heading. Vehicles can also be tracked by using optical systems like laser scanners. The location information obtained from these devices is however rarely used in terminals as such. Most automated terminals have a coordinate system, which has its origin tied to a corner of the terminal area, or some other fixed location. [10] This terminal-wide coordinate system will be further on referred to as terminal coordinates or global coordinates.

An object's position in the terminal is defined by its distance from the global origin along X- and Y-axes. An object's heading is defined as the angle between the object's X-axis and the global X-axis.

Control systems use their own internal coordinates, which may differ from the terminal coordinates in terms of origin and directions of the coordinate axes. Figure 10 illustrates this situation for two ASC blocks. The X-axis of each block runs along the gantry direction of the ASC cranes, whereas the Y-axis represents the trolley direction. One of the blocks has its trolley direction mirrored, which results in flipping of the Y-axis to the opposite direction.

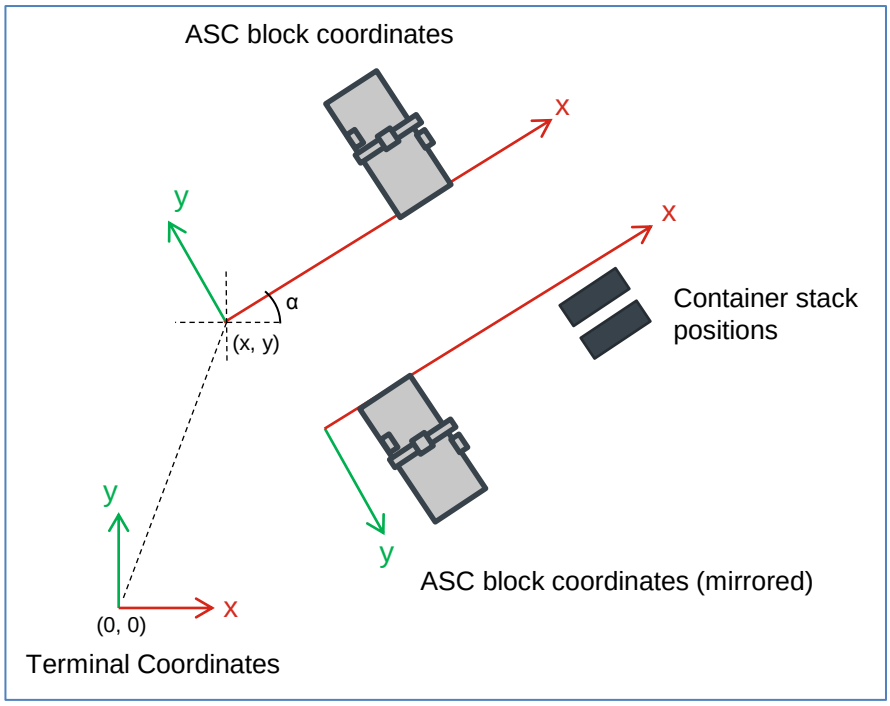

Figure 10: *Relationship of ASC block coordinates to the terminal coordinates*

Let (x_{block}, y_{block}) denote the origin of a non-mirrored ASC block. Assuming that both coordinate systems use similar units, the position of a crane (x_{crane}, y_{crane}) is given in terminal coordinates by

$$\begin{cases} x_{crane\_global} = x_{block} + (x_{crane} * \cos\alpha - y_{crane} * \sin\alpha) \\ y_{crane\_global} = y_{block} + (x_{crane} * \sin\alpha + y_{crane} * \cos\alpha) \end{cases}, \quad (1)$$

where α is the angle between the global and local X-axes. Calculations like this are performed also, when the container stack positions are converted into terminal coordinates. For HT equipment the situation is simpler, as the HTCS typically uses terminal coordinates without variations.

3.3.2 Configuration files

Fleetview uses several own file types for configuration. The most important ones considering the 3D visualization tool are *terminal_layout.xml* and *machines.xml*.

Program 2 presents a simplified example of *terminal_layout.xml*. This file type is used to describe the static terminal layout to the application. The example layout contains a water area and a building, which are defined as polygons. The fence and the gate are defined as linear structures, limited by two end points. Other objects commonly listed in *terminal_layout.xml* include rails, light poles, truck lanes, traffic lights, truck driver kiosks and various labels that are used to mark certain areas in the map.

```
<?xml version="1.0" encoding="UTF-8"?>
<Terminal width="2000" height="1000" >
    <Sea points="1000,800 1000,1000 2000,1000 2000,800"/>
    <Fence points="100,100 100,900"/>
    <Gate id="EXAMPLE_GATE" points="100,500 100,508" />
    <Building points="200,400 200,406 206,406 206,400"
           text="Example Building" />
</Terminal>
```

Program 2: *Example of terminal_layout.xml*

The machines and control systems of the terminal are listed in *machines.xml*. An example of the file syntax is given in in Program 3. The example script describes one ASC block with two cranes and one straddle carrier controlled by a HTCS.

The *machine* nodes of the example contain following attributes:

- *name* specifies the name for the machine, which is shown to the user. The naming system varies from terminal to terminal.
- *peerName* specifies the UniQ peer name of the machine. It is used for establishing communication between the application and the actual CHE. This attribute is made separate from *name* to maintain standardized naming convention between TLS applications.

- *controlPeerName* links the machine to the corresponding control system. This information is used e.g. in coordinate mapping.
- *type* specifies the type of the machine. This information is used not only for visualization but also for creating the corresponding data structures in the back-end.
- For the ASCCS, a block coordinate system is defined, like demonstrated in the previous section. The *heading* attribute defines the direction of the local X-axis, whereas the Y-direction is specified according to the *mirrored* attribute. The size of the cranes is given by *railWidth* attribute.
- ASC cranes have their own *mirrored* attribute, which defines the orientation of the crane on the rails. Initial position of the crane is given in *maintenancePosition* attribute, which is given in local coordinates.
- The SC has a zero position and default heading, which are used for setting initial position for the visualized machine.

```
<?xml version="1.0" encoding="UTF-8"?>
<machines>
    <machine name="HTCS"  peerName="HTCSM"  type="HTCS"/>
    <machine name="SC001" peerName="SC001M" type="Straddle"
           controlPeerName="HTCSM" zeroPos="0,0" heading="90"/>
    <machine name="ASCCS001" type="ASCCS" blockName="B001"
           zeroPos="700,300" heading="180"
           mirrored="true" railWidth="25.603"/>
    <machine name="ASC001W" peerName="ASC001WM" type="ASC"
           mirrored="true" controlPeerName="ASCCS001"
           maintenancePosition="300,0"/>
    <machine name="ASC001L" peerName="ASC001LM" type="ASC"
           mirrored="true" controlPeerName="ASCCS001"
           maintenancePosition="10,0"/>
</machines>
```

Program 3: *Example of machines.xml*

The configuration files are parsed during startup of Fleetview and the 2D view is initialized according to the data. A tag facade is initialized in the back-end and a connection is established to a Spread daemon, whose address is specified in another configuration file. UniQ channels are established between Fleetview and the peers listed in *machines.xml*. Finally, tags, alarms and events are requested from the remote peers. When ordering frequently changing tags, such as CHE positions, the amount of transferred items per second is limited to avoid excessive CPU and network usage.

The presented concepts of UniQ communication, data tagging and XML configuration files represent the input data and communication interfaces for the 3D visualization tool. The implemented application should work analogously to Fleetview, taking into account the requirements arising from the three-dimensional view. These include the visualization of objects' height and animation of movements in three dimensions. The formal requirements analysis for the application is performed in Chapter 5. Before discussing the requirements, an overview of existing 3D applications is given in the following chapter.

4. EXISTING 3D APPLICATIONS

In this chapter, a selection of existing 3D applications is presented in order to get an overview of what has been implemented so far and what such tools can accomplish. The first subchapter presents the 3D tools used in Kalmar, including the existing simulators. In the second subchapter, virtual container terminal solutions by other companies are discussed. In the third subchapter the focus is moved from the application domain to the use of game engine as the implementing technology. A literature review is performed of previously done projects that use game engines for simulation and/or interaction with real-world processes.

4.1 3D tools used by Kalmar

Various software tools have been used by Kalmar to simulate operations of machines since the middle of 1990s. The first applications to provide 3D animations were CAD and FEM (Finite Element Method) systems. These tools were – and still are – used to visualize parts and assemblies. In the 2000s more advanced tools were introduced, which feature multi-body dynamics simulation. These tools provide the possibility to simulate behaviour of complex machine systems off-line, i.e. the calculation doesn't happen in real-time but the results may be replayed at any desired speed. [22]

The next step in the evolution of simulation systems were the real-time machine simulators, which allow visualization of machine behaviour from various perspectives, e.g. from the viewpoints of driver and external viewer, parallel to the simulation. These systems also allow including actual control software and/or hardware in the simulation, which makes them important tools for product development projects. The visualization tools for creating the 3D view are usually included in the simulator software. [22]

One example of real-time machine simulators is the straddle carrier simulator, which is illustrated in Figure 11. The system replicates a physical driver's cabin of a SC that is working in a virtual terminal. The 3D environment is visualized on 6 screens around the driver, and the cabin mock-up is mounted on a motion platform for improved immersion. Another example is the RTG simulator, which is illustrated in Figure 12. The virtual RTG crane is operated via a remote control table, which is similar to ones used in actual terminals. The system visualizes the crane from various angles and has an onboard EMS-application running on a separate screen. Both of the mentioned systems utilize actual control algorithms, actual control hardware, UniQ communication and GUI components, whereas the 3D visualization is generated by the software of the simulator system provider.

Figure 11: *Straddle carrier simulator at Kalmar Technology and Competence Center, Tampere [23]*

Figure 12: *Operating of an RTG simulator via a remote control table [23]*

In recent years, the development has led to using of game engines for visualizing simulations of larger systems. Game engine based visualizations have already been created for terminal productivity simulations. These simulations use simplified control algorithms, which mimic the behaviour of TOS and other control systems. [22]

Another emerging area, where game engines have a significant role, is the virtual reality (VR) applications. These applications include a head-mounted display, which allows the user to be fully immersed into the 3D world, moving and rotating the view according to the user's head position. In Kalmar, experiments have been done of using Unity game engine to create virtual terminal environments for Oculus Rift headsets. [22]

In addition to simulators, 3D technologies have already been utilized to some extent in GUI development. UI application named DreamPort provides a complete set of tools for monitoring and operating a terminal that uses automated straddle carriers (AutoStrad) for container handling. The 3D GUI connects directly to the real-time control system of the AutoStrad platform, used SC terminals, which don't have the whole TLS software stack installed. [17; 24] Figure 13 shows an overview of a terminal area in DreamPort. SC operation is illustrated in more detail in Figure 14.

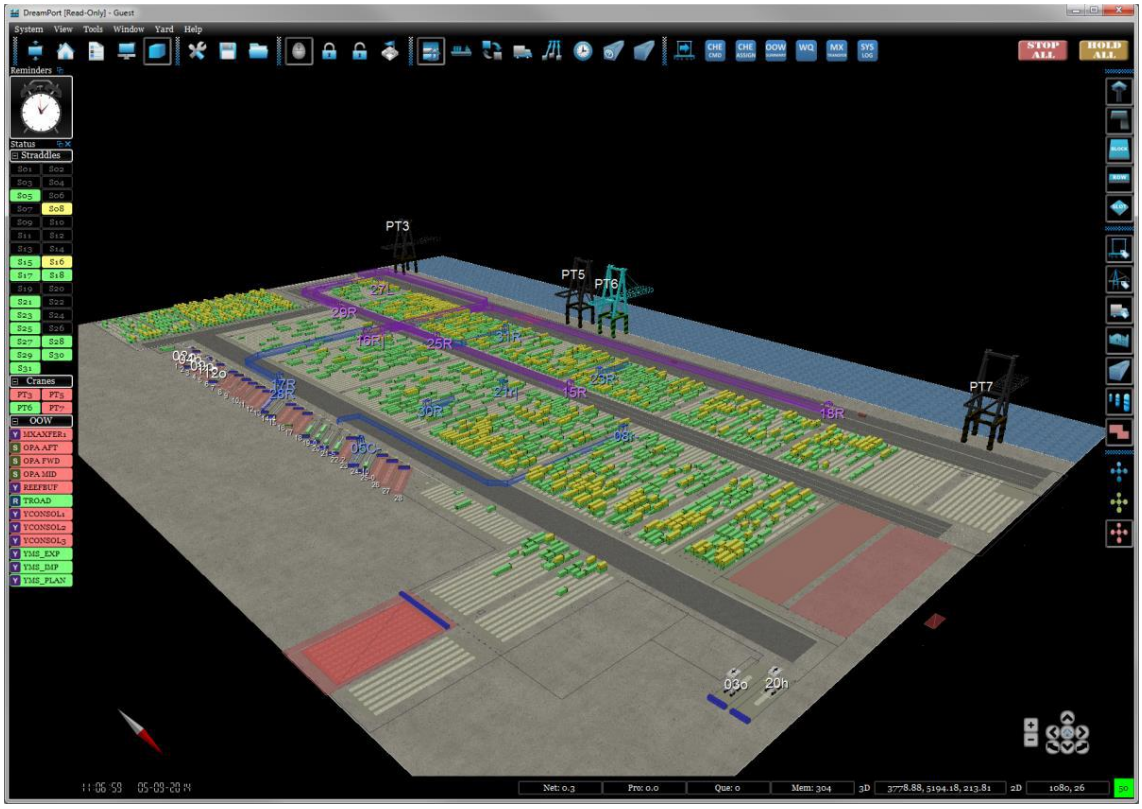

Figure 13: *Terminal overview from DreamPort [24]*

DreamPort features a fully navigable 3D view of terminal area, along with containers and machines, whose statuses are updated in real-time. Various colour coding systems can be applied to objects e.g. to highlight straddles that are working for certain STS or

to colour the containers according to their estimated departure times. Various other views are provided for terminal supervision and managing tasks. [24]

Since using of DreamPort is restricted to AutoStrad terminals with a certain type of control system, it cannot be considered as a complete 3D substitute of Fleetview. The latter is built on generic GUI framework and it bases its functionality on UniQ communication instead of being tied to a certain control system. That makes Fleetview more versatile in visualizing different types of terminals, albeit only in 2D.

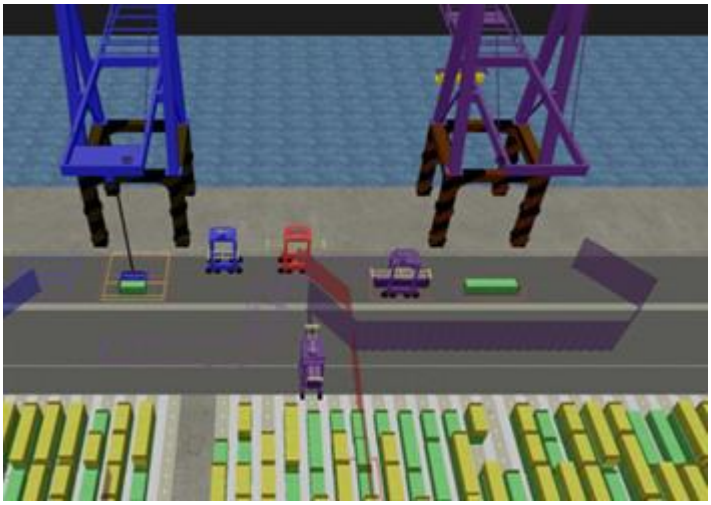

Figure 14: *Straddle carrier operation visualized in DreamPort [24]*

4.2 Virtual terminal solutions by other companies

Applications that allow 3D visualization and/or simulation of functional terminals have already been developed by several companies besides Kalmar. These companies are typically not providing physical harbor equipment themselves, but they have focus on container terminal software or simulation tools in general.

FlexTerm is container terminal simulation software built on FlexSim simulation engine. It can be used for modeling, visualizing and analyzing a whole terminal or a single activity, like gate operation. FlexTerm library provides ready-made 3D models for all common CHE types, vehicles, vessels and fixed structures like rail tracks. Results of the simulations can be analyzed graphically through 3D animation, and through statistical reports and graphs. The software can be used for all the planning activities described earlier in Chapter 2. [25; 26]

Terminal View by Tideworks is a comprehensive toolset for on-line visualization of terminal operations. It features a freely navigable 3D view of the terminal area, based on realistic models of containers, vessels and CHE. In addition to real-time visualization of operations, the GUI provides search tools for easy location of a certain container or CHE. Color coding and filtering can be used e.g. to find containers holding hazardous

materials or to find equipment, containers and stack positions associated with certain operation. In addition to the general 3D view, the application provides several operational views and analysis tools for reviewing the situation in the terminal. [27] Since 2013 Terminal View has been separated from the TOS and other control applications provided by Tideworks. The application provides visualization for data that is obtained from external systems, which may include third party applications. The 3D view is created by using a graphics engine from the gaming industry. [28] Technical details of the visualization are however not revealed by the company.

The CHESSCON software suite by ISL Applications provides a range of tools for planning, simulation and visualization of terminals. CHESSCON Terminal View is a planning application, which allows easy creation of 3D terminal models from layout drawings, which are created in 2D editor. The application provides a library of 3D objects, which can be used alongside user-imported 3D models. CHESSCON Yard View is an operational tool for storage yard planning. The application communicates with TOS and generates a real-time 3D view of the container inventory, which can be filtered according to user preferences. [29] CHESSCON Virtual Terminal is a terminal emulator, which can be combined with an existing TOS and further software systems to create a virtual copy of an existing or not yet implemented terminal. The application contains an internal logic for handling the equipment and it provides emulation models for the most common CHE types. Using of external CHE emulators and 3D models is also possible. The application generates 3D animation of the terminal events, which can be played at any speed. Further views are provided for on-line monitoring and off-line evaluation of operations. [30] In addition to the mentioned tools, the CHESSCON suite includes three further applications for various simulation and planning tasks of a terminal. [29]

The Netherlands based company TBA provides control systems for automated container terminals, along with simulation tools and consulting services. TBA's CONTROLS (CONtainer TeRminal Optimised Logistic Simulation) is a virtual terminal application designed to be used with actual TOS. The software emulates the control systems and physical processes of a terminal, which is realized by using internal simulation models. While using CONTROLS, the TOS can be configured and tested and operators can be trained without affecting the real operations. The application provides detailed 3D visualization of the operations, which can also be replayed when desired. [31] The visualization component used in CONTROLS was developed using Ogre3D, which is an open source game engine based on C++. The application uses a database for storing the events, which allows for replaying the events or viewing them on a remote computer. [32]

Another interesting concept by TBA is the Safe-T Game, which allows the users to be immersed in the virtual terminal as game characters. The game system utilizes TBA's equipment control system TEAMS, which is running live operations on actual CHE. The virtual representation of the terminal is created by CONTROLS software, which

replicates the events of the actual terminal in the game world. The players move in the terminal in the roles of dispatchers or maintenance technicians, facing various scenarios. For the visualization, Oculus Rift headsets may be used instead of conventional computer displays to achieve highest possible level of realism. The game allows training of personnel based on actual situations in the terminal, thus raising the general awareness of the risks in the automated terminal environment. [33; 34]

A few further papers have been published about virtual terminal applications, which are not directly linked to any of the previously mentioned commercial products. Lau et al. [35] describe a design for a distributed real-time training environment, which consists of a STS simulator and a simulated control tower. The crane cabin simulation is running in a CAVE (Cave Automatic Virtual Environment) like environment, where the view is projected on multiple screens around the user. The control tower operator is supposed to give instructions to the crane operator according to the loading plan, which is visible in the control tower UI. The movement of the crane is in turn visualized in the control tower view, allowing realistic cooperation between the operators.

Bruzzone et al. [36] describe design processes of a straddle carrier simulator and a truck simulator, which can be used for training purposes. A virtual terminal environment is created based on an actual major terminal in southern Italy. Several technologies and modeling tools are utilized in the project, including 3D Studio Max for physical modeling of objects, Microsoft Visual C++ for simulating the behaviour of the straddle carrier, Quest 3D for implementing the truck simulation and Vega Prime for visualizing the terminal environment. The SC simulator provides multiple views of the machine, including cabin view, external view and bird-eye view. The simulator is operated via a physical steering wheel, pedals and joystick. The truck simulation is linked to a weather control system, which can be used to create various driving conditions for the simulator. The truck is controlled via steering wheel and pedals, and the simulator system features multiple LCD screens to provide a realistic view. Both of the implemented simulators feature automatic mission control systems, and they also allow for evaluation of the driver performance.

4.3 Utilization of game engines in other fields

In this section, a selection of previously implemented 3D applications from fields outside container handling industry is presented. In order to narrow down the scope of the study, the example cases were chosen according to the following criteria:

1. The application is implemented by using a game engine, and
2. The application communicates with an external process, or it is used in an interactive manner to emulate events of the real world.

Based on the criteria, about 40 relevant cases were found with topics ranging from architectural planning to physics modeling and various learning simulations. The applications referenced in this section represent approximately one third of all the studied material. The cases that were left out were either relatively old or they were considered uninteresting or very similar to the cases already cited.

The term serious game is commonly used to represent games that are developed for other purposes than pure entertainment. Common fields of application for serious gaming include military, healthcare, education, virtual cities, and different kinds of training simulators used by companies. These simulations don't however necessary have any interfaces to the external world, nor are they always implemented by using a commonly available game engine.

Yuan et al. [37] present a design for a simulated training system for firefighters. Their paper describes a distributed training environment, consisting of an application server, a database server and multiple PC workstations, which are used to simulate different roles of the fire crew. The fire scene is presented in a 2D map representation and an interactive 3D scene implemented with open source game engine Delta3D.

Chittaro and Ranon [38] take a different approach to the fire protection. They describe a serious game for providing virtual evacuation training for civilians. The presented evacuation game creates realistic scenarios of fire situations in a modeled university building. The scenarios are organized into levels with increasing difficulty, and the game features a scoring system to evaluate the player's performance. A separate 2D tool can be used for analyzing the movement of players after the game. The game is implemented with NeoAxis game engine, while using application named 3D Studio Max for modeling the objects.

Mól et al. [39] describe another application for evacuation simulations. Their solution is implemented by using Unreal Engine, and it provides a distributed environment with one supervisor and multiple players. In addition to the ordinary computer screens, the 3D scene can be visualized on a 2 m x 3 m stereo-projection screen. The authors suggest that the VR approach could be used beneficially to simulate emergency situations in hazardous environments, like nuclear power plants.

In [40] an interactive space walking simulation for Oculus Rift headsets is created using Unity as the game engine. Pictures and 3D models from NASA archives are used to create realistic models of the Moon's surface, Apollo Lunar Lander and the International Space Station. Physics engine of Unity is used to simulate gravity conditions of actual locations, while the controlling of player characters is implemented via conventional pad controllers. The application is supposed to serve as an educational tool in an observatory environment.

Kaasalainen [41] describes the use of VR technologies in rehabilitation and perception analysis. The prospective users include stroke patients and other people with reduced physical abilities. Two applications are created by using Unity for programming and Oculus Rift for visualization: OcuTread simulates moving in a virtual outdoor environment for a user walking on a treadmill. Purpose of the application is to promote walking training for stroke patients, who have temporarily lost their motoric abilities. The application uses pedometer software of a smartphone to measure the walking speed of the patient and to translate the corresponding movement to the virtual environment. OcuCar is a driving simulation, which includes a steering wheel controller with force feedback functionality. Besides driving, the user is supposed to react to signals generated by the application by pressing the push buttons on the controller. The application is supposed to measure changes in reaction times due to driver fatigue, and on elderly patients.

In [42] a driving simulator is developed, which integrates a steering wheel and a set of pedals into a simulation system consisting of a motion platform, a driver's seat and Oculus Rift headset. The application is developed using Unity, and its purpose is to provide a reusable set of simulation components, which can be combined with any upcoming Unity projects. The project is expected to be used as a basis for ambulance driving simulations, which would serve as part of virtual learning environment for healthcare students.

Further training simulations for medical staff are discussed in [43] and [44]. In the first paper, three game engines are evaluated in order to find the best possible platform for creating low cost clinical training applications. The implemented simulators should be built by using custom content and they should allow cooperation between multiple users over network. Sample applications are built for all three engines. The featured engines are however relatively old from today's perspective. The latter paper describes a surgical cutting simulation based on deformable objects. The implemented application allows arbitrary cuts in the objects, proving that low-cost game engines could be used to simulate complicated surgical procedures.

In addition to serious games, 3D simulation visualizations are commonly used in design tools of industrial systems. Common fields of application include robotics, architecture, flow simulation, and design of power plants, just to name a few examples. These design tools are often based on proprietary technologies, and their implementation details are rarely revealed by the providing companies. Some of the reviewed cases describe design processes for tools, which serve as in-house substitutes for commercial applications. This underlines the fact that modern game engines allow relatively easy creation of tailor-made visualization tools, given that the designers have sufficient knowledge of 3D models and software development in general.

Korkiakoski [45] describes the use of Unity and VR technologies for visualizing three-dimensional building models. The application is built on an existing software frame-

work, which creates 3D models from building layout drawings. A browser application is used to visualize the models and to modify them with different surface materials and furniture setups. The VR application adds a further level of immersion to the framework by allowing virtual walking inside the model. In the thesis work, two implementations are created for Oculus Rift and Google Cardboard platforms.

Wang et al. [46] describe a software package for visualizing industrial processes in Unity. The paper presents three example processes from iron and steel industry: blast furnace, overhead crane and vertical edger. Separate simulation software is used to perform computational fluid dynamics and finite element analysis on the simulated processes. The simulation results are imported into Unity, which uses predefined 3D models to construct virtual environments based on the simulations. The application allows direct interaction between the user and the environment, simulating the operations of a real factory. The simulated cases can be used as training scenarios, which are especially useful for emulating extreme working conditions. Implementations of the visualization software are built for various platforms, including immersive theater, conventional PC, smartphone and Oculus Rift. According to the paper, the physics model of the game engine would not allow sufficiently accurate simulation to be viewed with a realistic frame rate. The presented application uses precomputed simulation data, which makes it essentially a non-real-time simulator.

In [47] a real-time monitoring system for a production line is developed using Unity. The featured example considers a single workstation of a production line, consisting of a working robot and conveyors, which are used for transporting pallets. The workstation communicates with the web-based data acquisition system by using XML-based messages. A 3D model of the workstation is created using CAD/CAM software and exported to modeling application Blender to make the model compatible with Unity. The monitoring tool is implemented as a web-based application, which is used via Unity browser plugin. The application provides animation for the robot and the pallets according to the process data.

Makkonen et al. [48] demonstrate the use of open source tools for creating 3D visualizations for mobile machines. The study features a machine control system of an excavator, which is visualized with a free open source game engine Panda3D. The application uses GPS data of the actual machine to visualize the excavator on a virtualized roadwork site. The authors suggest that the control system data of machines should be more commonly distributed by open methods to promote new ways of benefiting from the existing data.

Korpioksa [49] describes the using of programmable logic controllers (PLC) in cooperation with Unity. Three different PLC systems are used to send commands to a virtual drilling workstation, which is implemented with Unity. The Unity application then sends simulated return values to the PLC system. Response times are obtained for two

out of the three setups. The author states that some 3D modeling tools that are commonly used with PLC systems lack the real-time physics simulation, which is included in Unity. This would make Unity a feasible choice for developing virtual test environments for PLC applications.

Patana [50] writes about the design and implementation of simulation and programming tool for abrasive blasting robots. User of the application is able to load a robot and scene model from a file and create work instructions for the robot by adding target points in the 3D space. The resulting program can be visualized in the application or saved to a file and uploaded to an actual robot. The application can be considered as an alternative to the existing commercial robot simulation environments with similar functionality. The data model of the application is written in Qt and the 3D visualization is generated with Kajak3D game engine, which is developed in Kajaani University of Applied Sciences.

According to the performed literature survey, experimental projects utilizing game engines have already been done for a wide range of applications. Some of the reviewed cases include small scale integration of physical machines or simulation systems to the 3D environments. There are also commercial tools available which allow connecting a whole container terminal – either a physical one or simulated – to a real-time 3D visualization. Some of these, namely TBA CONTROLS and Tideworks Terminal View, are proven to use game engine technology for generating the 3D view. The rest of this thesis describes development of similar solution based on previously discussed Kalmar products.

5. REQUIREMENTS FOR THE VISUALIZATION TOOL

In this chapter, requirements analysis is performed for the terminal visualization tool. In the first four sections the application is defined in terms of what it should do. These requirements were obtained based on the initial goals of the client, and the formal requirements specification was written in cooperation with the client during the first weeks of the thesis project. The resulting functional and non-functional requirements are translated into technical requirements in section 5.5. This analysis serves as a starting point for the game engine comparison, which is performed in the following chapter.

5.1 Purpose of the application

Purpose of the application is to provide a smoothly flowing and interactive 3D visualization of a virtual container terminal, which is being simulated on a separate server. The involved CHE, containers and their environment should be modeled with such a precision that it creates a believable illusion of a working terminal. The application shall be implemented by using a modern game engine framework, which is able to use imported 3D models.

5.1.1 Intended use

This application, known as *FleetSimulator*, may be considered a sub-project to a larger virtualization project within Kalmar. The ultimate goal is to create entirely functional virtual terminals prior to construction of any actual terminal equipment. This goal would be achieved by using actual terminal software, including TOS and control systems, and simulation models for CHE. Besides CHE and containers, the application should be capable of visualizing the static terminal layout, including apron, sea, buildings, rails, fences and further objects.

Possible users of the application include marketing people, who could use the application for demonstrating Kalmar solutions to prospective and existing customers. It could also be used by software developers, testers and project engineers to visualize the systems they are working on.

Including the application in customer software packages or using it for on-line visualization of actual equipment is not in the scope of the project. Although it should be tech-

nically possible as the application shall utilize the same communication principles that are used in production systems.

5.1.2 Features to be included

Since the application is supposed to utilize external simulation models, its focus should be strictly in visualization. No physics modeling or trajectory planning should be performed in the game engine platform, as the location and trajectory information for objects is acquired real-time from the simulation environment. The data acquisition rate might however be variable or too low for creating an animation smooth enough to appeal human eye. Thus, the application shall use the necessary methods to smoothen the movement for the 3D view.

CHE status monitoring is beyond scope of the application, since efficient software tools already exist for that purpose. The application could be run in conjunction with Fleetview or EMS to allow both extensive monitoring of CHE and 3D visualization of the events in the terminal. The possibility to create links from the 3D GUI to other tools may be studied, but the application shall also be able to run independently without need for any further software installations.

5.2 Application environment

The simulation environment for *FleetSimulator* should use actual TOS and Control System software, whereas the machine layer would be realized by simulation models that represent the CHE on the field. The assumed use case for the application is illustrated in Figure 15. The simulation environment – which is described on a very abstract level – could theoretically be replaced with actual TLS and the situation to *FleetSimulator* would remain essentially the same. Some aspects of the presented use case are covered in the following sections.

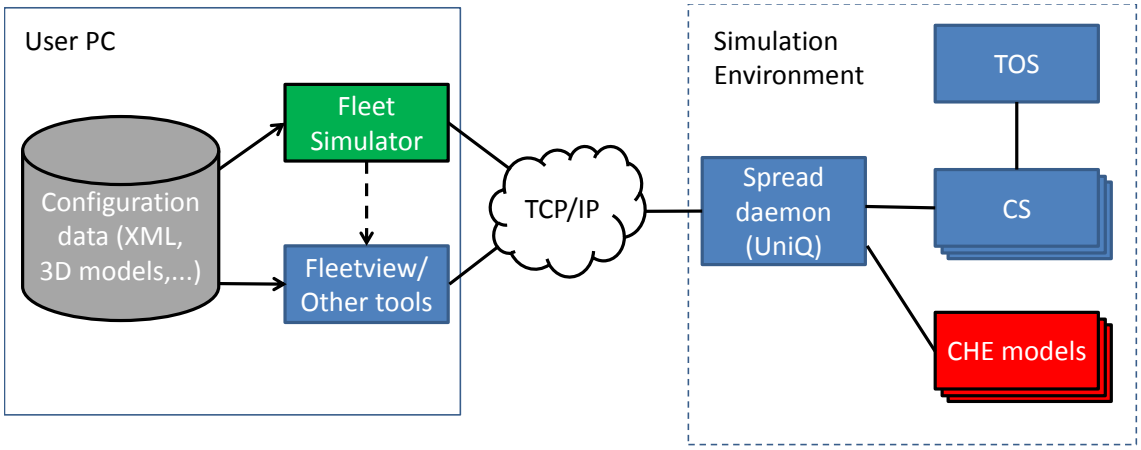

Figure 15: *Expected use case for the application*

5.2.1 Connection to the simulation environment

The communication between the application and the simulation environment is established by means of UniQ platform. The communication channel is built by connecting to the Spread daemon, which is running on the simulation server. The daemon may be accessed by using standard TCP/IP communication, thus the only parameters for the connection are the IP address and port number of the daemon. These are combined to a connection string, where a string *4803@127.0.0.1* would represent a spread daemon running on the local host.

The most likely use case involves the application running on a PC, which is connected to the simulation environment via local network or over the Internet with VPN. This situation was illustrated in Figure 15.

5.2.2 Configuration data

In addition to the dynamic data from the simulation environment, some static data is required for building a realistic model of the terminal. The application shall be able to utilize same configuration files that are used by Fleetview, most importantly *terminal_layout.xml* and *machines.xml*. Further XML files are required for UniQ communication, including *tagmap.xml*. Without these configuration files the visualization of the terminal and communicating with TLS would not be possible in the first place.

To enhance the reusability of the XML data and the genericity of the application, as little data as possible should be hard coded in the application. Run-time XML parsing and procedural object generation should be used instead. It is although not absolutely necessary to import the static terminal layout in order to visualize CHE and containers. A blank asphalt field and some ready-made layouts may be provided as alternatives to the dynamically created layout.

Realistic 3D models of objects are also required for visualization. It should be possible to import such models in some common 3D modeling format, either in runtime or in the design environment. In the ideal case, adding and replacing of models should be possible without making changes to the application code.

Configuration data shall be organized in a parameter folder and the application shall provide a mechanism for defining the parameter folder address, along with other necessary input parameters.

Creation and acquisition of mentioned configuration data is beyond scope of this thesis. Users of the application must take care that the configuration files and required 3D models for each project are available, valid and properly installed on the system.

5.3 Functional requirements

Functional requirements of the application are divided in three priority classes. The features listed as having a **high priority** represent the essential functionality to be implemented in the prototype. The **medium priority** features supplement the core functionality and their implementation is dependent on available time and resources as well as the difficulty of the implementation. The **low priority** part lists additional features, which will likely not be implemented in the prototype. The possibility to include these features shall however be taken into account in the application design.

5.3.1 High priority

R 1 The application shall visualize the main components of the CHE based on the predefined 3D models and dynamic status information from TLS. Movements of the CHE shall be animated at a rate resembling real-time motion. The prototype shall provide visualizations for ASCs and automatic straddle carriers.

R 2 The application shall visualize containers and their movements according to the dynamic information from TLS. Handling of containers shall be animated smoothly together with the CHE. The prototype shall provide visualizations for most commonly used general-purpose containers with a few different exterior options.

R 3 The application shall provide a 3D view of the terminal environment, which can be zoomed and rotated freely.

R 4 The application shall be able to import and visualize the static world model from an external source, such as *terminal_layout.xml* or a previously created 3D-model in .blend, .fbx or .3ds format. As an alternative to these, a default world model consisting of a blank asphalt field should be provided.

5.3.2 Medium priority

R 5 The application shall visualize routing information for the CHE.

R 6 The application shall visualize space reservations for the CHE and the target slots for the carried containers.

R 7 The application shall visualize equipment ID.

R 8 The application shall allow communication from the GUI to the simulation environment. E.g. the pressing of a push button on a truck kiosk shall toggle corresponding communication.

R 9 In addition to the freely navigable 3D view, the application shall provide certain standard views, which can be toggled at any point by using a hotkey or visual menu. These standard views shall include at least a top view and isometric views from four main directions.

5.3.3 Low priority

R 10 The application shall have a configuration view, where the source of static world model (e.g. Default/XML/Other) and connection details to dynamic data source are specified. This configuration shall be done once during initialization of the application. If no such view is provided, a dedicated configuration file shall be used instead.

R 11 The application shall have a preferences view, where certain visualization features - like route and/or reservation information, can be toggled on and off.

R 12 The application shall allow controlling of CHE from the game environment. E.g. moving and lifting commands may be implemented for some CHE types.

R 13 The application shall provide machine views, where the camera can be locked on a certain CHE or some other object to track their motion from different angles.

R 14 It should be possible to get the status information for a certain CHE by sending commands to external applications (e.g. EMS or Fleetview).

R 15 The GUI shall feature a machine list, which can be used for locating certain CHE and toggling functions specified in R 12 – R 14 for the selected CHE. The CHE could also be selected in point-and-click manner.

R 16 Adding new 3D models or replacing existing ones should be possible without need for modification, recompilation or reinstallation of the application.

5.4 Non-functional requirements

R 17 The application shall be implemented as a stand-alone executable. No installation of other applications shall be required and the installation package shall contain all the necessary libraries and other resources.

R 18 It shall be allowed to create and run a.m. executables within Kalmar without any payment to game development application supplier.

R 19 The most commonly used 3D models and XML configurations shall be included in the installation package.

R 20 The application shall be usable on a gaming level PC running a 64-bit Windows 7 or newer.

R 21 Connection to the dynamic data source shall be established via local network or a broadband Internet connection with VPN.

R 22 The application shall be able to smoothly visualize terminals with up to 60 CHE and 10,000 containers.

R 23 Containers shall be handled as either static or active objects according to their role. A container to be transported shall be converted to an active object and again to a static object once it is grounded.

R 24 Data update interval from TLS shall be sufficiently short to allow smooth visualization and sufficiently long to prevent excessive CPU and network usage. The interval may be adjusted according to terminal size to meet these criteria.

R 25 The application shall utilize a world coordinate system similar to those used in the underlying models.

5.5 Technical considerations

The previously listed functional requirements lead to certain technical constraints, which act as guidelines for the game engine selection process. When comparing to the traditional view of game development and the design process commonly presented in handbooks and tutorials, our application has certain distinctive features:

- Instead of building the 3D environment in editor before compiling and running the application, nearly all objects have to be instantiated run-time according to the XML data. The instantiation shall be controlled by the application code and the solution must scale to any number and combination of objects as long as their types are known.
- Instead of defining object behaviour in scripts, the movement of CHE and containers should be based on the external data from TLS. As the number of CHE or control systems is not limited, these behaviours need to be copied along with the visible 3D objects.
- The application shall animate the state of the objects at a frame-to-frame rate regardless of the actual data update rate from the simulation environment. This suggests that the data acquisition should be separated from its usage.
- The GUI shall also be able to interpret user inputs and send messages back to the simulation environment.

Due to these assumptions, the chosen game engine should be relatively flexible in terms of programmability: Run-time instantiation of objects should be straightforward and the interaction between objects should be easy to implement. In order to make the GUI run independently from the data acquisition, some type of multithreading is required. As the UniQ communication shall be achieved by using ready-made libraries, the engine should also be able to include external DLLs in the solution.

5.5.1 Proposed architecture

The dynamic nature of the application and the amount of required features calls for a considerable amount of programming. In order to make the game engine project as simple as possible the data acquisition should be implemented in an external library. Figure 16 presents the proposed architecture for the application.

In the presented architecture, the game scripts are solely responsible for creating and controlling the visible objects of the application. Data acquisition and storage is implemented in a back-end library, which consist of several functional blocks:

- **Object cache** is responsible for providing common storage types for objects and storing the data in such a form, that it can be easily consumed by the game scripts.

- **XML reader** contains the logic for parsing the configuration files and adding the static objects and machines to the cache.
- **Communication manager** is responsible for setting up the UniQ communication and converting the arriving tag values into updates in the cache (and vice-versa). After the initialization, the UniQ communication shall run in a separate thread in order to decouple the data acquisition from its usage.
- **Session handler** controls the program workflow by passing calls to other blocks.

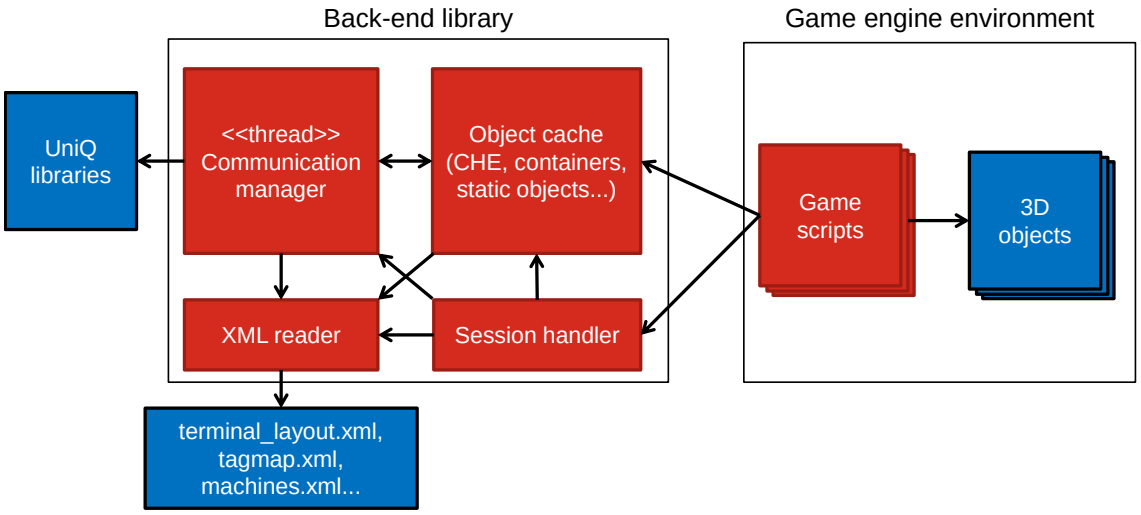

Figure 16: *Proposed architecture for the application*

The presented architecture has some obvious advantages: As the required information is stored in the cache, the data model can be tailored to meet the needs of the game scripts. Coordinate mapping, unit transformations and transitions from 2-dimensional to 3-dimensional space may be performed in the back-end, and the game scripts may always receive the exact 3D world positions of the objects.

The 3D models are only known to the game engine project and the game scripts may concentrate on propagating the changes in the cached model to the visible objects (and vice-versa). The game engine project, on the other hand, is unaware of the communication manager and its dependencies. The game engine project depends only on the session handler and the common data model, which makes it possible to replace the whole back-end if necessary. The back-end should also be independent of the used game engine technology.

As the game engine project and the back-end are separate software entities, their development and maintenance can be done separately from each other, as long as the specifications and public interfaces are kept unchanged.

6. CHOICE OF GAME ENGINE

Purpose of this chapter is to provide a deeper insight of what is understood as a game engine and to perform a comparison between two engine candidates, one of which was chosen as the development platform for the visualization tool. General characteristics of game engines are discussed in section 6.1. The engine candidates for the visualization tool development are presented and compared in section 6.2. Conclusions from the analysis are presented in section 6.3.

6.1 Functions of a game engine

Finding an unambiguous definition for game engine is not a simple task. The term is commonly used to describe a software framework or toolset, which allows simplified development and execution of video games. More concretely, a game engine provides game developers with tools that allow them to concentrate on creating the game content, while letting the engine take care of lower level technical details.

Typical functions of a modern game engine include the following [51-53]:

- **Rendering system** is responsible for generating the graphical view and converting the calculated graphics into corresponding draw events on hardware. It is one of the most fundamental parts of an engine, since it brings the defined game world objects into visible pictures on the screen.
- **Scripting system** provides the tools for defining the game logic and adding intelligence to game objects via programming. The used programming languages and development tools vary from engine to engine. Some engines are also providing library functions and pre-made scripts, which the developer can modify to use them in his/her application.
- **Input system** is used to handle user inputs, which are given with keyboard, mouse, touch screen or similar devices. In the game logic, events can be tied to abstract input names instead of hard-coding them to fixed input values. The resulting input table can then be mapped to actual devices by the end user.
- **Physics engine** is used to model physical behaviour of objects by applying concepts like gravity, friction and physical forces to them. Physics processing is typically separated from rendering, meaning that a visible 3D object may or may not have a physics body attached to it. The physics body may also have a different shape than the one that is being rendered. Physics modeling requires relative-

ly lot of computation power. Thus, it should only be applied to objects for which it is really needed. [52]

- **Audio system** handles sound effects and background music and converts them into corresponding audio events on hardware. A 3D game engine is also supposed to modify qualities of sound based on the offset between the listener and the sound source. Further effects may be added according to the environment, where the sound is being played. [52]
- **GUI system** is used to create and handle UI elements such as text boxes, buttons and in-game menus. If these functions are not provided by the engine itself, they may be written by the developer or imported as third-party libraries. [52; 53]
- **Networking system** provides tools and software components for handling the network communication in multiplayer games.

Implementations of game engines vary from reusable software libraries to fully integrated development platforms, with graphical level editors and design-time testing functionalities. Some engines are specializing in 2D or 3D graphics, while some can be used to develop both kinds of applications. The target platforms, where the developed applications can be used, are also varying from engine to engine.

In this thesis, the focus is on using services of a game engine to enable visualization of container terminals according to simulation data. Underlying concepts, like 3D graphics and physics modeling, are not covered in this document beyond the level that is necessary for understanding the problems at hand.

6.2 Candidates for the implementation

The comparison of development platforms for the visualization tool was made between two commonly used modern game engines: Unity and Unreal Engine.

Unity (also known as Unity3D) is a cross-platform game engine developed by Unity Technologies. The first version of Unity was released in 2005 as an attempt to provide amateur game developers with tools and technologies, which had previously been a privilege of large game companies. By 2013, Unity Technologies had expanded from a team of three programmers into a global software company, employing 285 people. [54] The latest major version of the engine, Unity 5, is capable of producing visually appealing 2D and 3D applications for a wide range of platforms.

Unreal Engine (UE) by Epic Games was first introduced when it was used in the 1998 game *Unreal*. Since the release of the first engine version, the company has been licensing the technology to other game companies so that they could use it in their titles. In 2009 a freely downloadable edition of Unreal Engine 3, known as Unreal Development Kit, was published. [55; 56] The latest generation of the engine is Unreal Engine 4,

which is offered with equal terms to independent game developers and large companies. It is widely used for developing 2D and 3D applications for various platforms.

The number of engine candidates was limited to these two, as the time frame of the thesis project did not allow for making in-depth analyses of more technologies. The two featured engines are also among the most popular in their field: In March 2016, the global market share of Unity was claimed to be 45% with more than 4.5 million registered users, while UE was coming in second with a 17% share [57]. The popularity of an engine correlates strongly with the amount of free support material that is available on the Internet. Since game engines are very comprehensive toolsets, learning to use a new engine from scratch would be relatively difficult and time-consuming without sufficient training material.

The two engines were compared in terms of their applicability to the given development task. Many features, which are typically interesting from game development point of view, were not considered – like the capability to produce advanced graphical effects. Both engines were considered adequate for providing the level of detail that is required from the terminal visualization. Multiplayer or networking features of the engines were not examined, as the connection to the UniQ environment was supposed to be implemented by using external libraries. Finally, little attention was paid to the possibility of porting the application to different target platforms. The requirements specification states that the application shall run on a typical PC with Windows operating system. Possible further platforms, which could come in question, include VR systems and mobile devices, which are readily supported by both engines [58; 59].

The comparison was originally performed in early 2016 and the decision was made at the beginning of April. The discussed features reflect the situation at that time, while more recent changes are mentioned separately in the text.

6.2.1 Licensing

In April 2016, Unity 5 was available in Personal and Professional editions: The Personal edition was offered free of charge, while the Professional edition was available for US $75 per month or US $1500 for unlimited license, with limited accessibility to free updates. Both of the editions included the engine and an editor with all the basic features. There were however restrictions on who is eligible to use the personal edition: Purchasing of the professional license was required from commercial entities, whose gross revenue or funding exceeds US 100,000$ per year, and from non-commercial entities with yearly budget exceeding the said amount. To the applications built with the free version, a non-removable splash screen was included, stating that the application is made using Unity Personal Edition.

By October 2016, a new Unity Plus license has been introduced between the Personal and Professional editions, and the payment schemes and revenue restrictions have been modified: The new licenses are only available through monthly subscription, and the number of "seats" per license can be managed more flexibly than before. There are also separate revenue restrictions for the Personal and Plus editions. However, none of the licenses forces the user to pay any royalties from the revenue made by using or selling the created applications. [60]

Unreal Engine 4, on the contrary, is completely free to download and use for all kinds of developers. The engine is also open for modifications, as the C++ source code of the engine is available for all licensees, which is not the case with Unity. A royalty payment of 5% is required from all gross revenue from the developed applications, as long as one of the applications is grossing more than US $3,000 per calendar quarter. Some exceptions to this rule, as stated in the end user license agreement, are "architect-created walkthrough simulations or contractor-developed in-house training simulators" and "non-interactive linear media". The latter includes movies and video clips that are created by using the technology. [59; 61]

Epic Games is also offering special enterprise services for customers that are using UE for other purposes than game development. These arrangements concern the amount of support that is given to the developers. They don't add any costs to the use of the engine itself, nor do they remove the obligation to pay royalties. [59]

Based on this information, it seems that Unreal Engine could be used for developing the visualization tool without charge, as long as the application is not distributed in exchange for money. In Unity, there is a license fee involved, when using the engine in Kalmar projects.

6.2.2 Object model

Each of the engines provides a versatile editor environment, where the **scenes** of the game are built from **assets**, like 3D models, and further components provided by the game engine. The scenes can also be compiled and tested directly in the editor.

In Unity, the scenes are built out of **GameObjects**, which serve as containers for further components. For example, a GameObject for a shipping container could consist of a 3D model (a **mesh** with textures) that describes the physical appearance of the object, a box collider that simulates the physical behaviour of the object, and a script, that adds program logic to the object. Furthermore, each GameObject has a **transform**, which describes the position, orientation and scale of the object in the game world. GameObjects can have parent-child relationships: e.g. the container object can have a child object named door and when the container is moved or rotated in the scene, the door object moves and rotates along with it.

GameObjects can be saved as **prefabs**, which act as templates for new GameObjects. E.g. when the previously mentioned container object is stored as prefab, each **prefab instance** that is created will have exactly the same features than the original container. Changes to the prefab will affect all the prefab instances. E.g. if one wanted to replace the container model with a more realistic one, more complicated meshes and colliders would be required for simulating the behaviour of doors and twistlock fittings. Once done in the prefab, these changes would be copied to all container instances in the scene. Changes to an individual prefab instance will not affect other instances. E.g. when the surface colour of one container in the scene is changed from red to blue, other containers stay red.

Unreal Engine's equivalent of GameObject is the *Actor* class, which acts as base class for all objects that can be placed in the game world. Each actor contains a hierarchy of *Components*, which define the content of the actor, similarly to the components in Unity. Actors can be specialized and extended to create objects with custom functionalities. E.g. an *Actor*, which is used to represent a player or an AI controlled object is called *Pawn*. *Character* is a subclass of *Pawn*, with further components for modeling a creature with an animated body. The developer is able to create his/her own actor types and components through inheritance. [62]

An actor with components may be saved to a Blueprint class, which is the UE equivalent of a prefab. Unlike Unity prefabs, Blueprint classes can be extended to create new types from the existing Blueprints. This simplifies the implementation of new features, while changes in one Blueprint class are automatically inherited by all its children. [62]

While UE allows a lot of flexibility in defining the object model, the system used in Unity is much easier to understand for a beginning developer. The lack of inheritance on prefabs can be seen as a minor disadvantage: As many CHE types share similar properties, it would be useful to create one common *Machine* class and derive the individual machine types from it.

6.2.3 Modeling support

Unity allows importing of 3D models in at least 9 different file formats. The supported types include portable formats like .fbx and .obj. and various proprietary file types used by 3D modeling applications. The former can be imported in the Unity editor as such, while the latter require the corresponding modeling software to be installed on the computer. For these file types, automatic conversion is performed during the import. [63] Among the supported modeling tools is the free open-source application Blender, which can be opened directly by double-clicking a model in the Unity editor.

Unreal Engine supports currently only .fbx and .obj formats for external models [62]. Thus, a 3D model of any other format must be exported manually to one of the two formats before importing it in Unreal editor.

Unity allows importing textures in 11 different file formats, while the corresponding number for UE is 8. Commonly used bitmap image formats like .jpg, .bmp and .png are supported by both engines. [62; 64]

In addition to importing, both engines allow creation of 2D sprites and 3D objects directly in the editor environment. The native modeling capabilities of Unity are however limited to a few basic shapes, like rectangular boxes, spheres and cylinders. UE allows creating triangular objects and editing model geometry in the editor, while Unity does not. More complicated shapes can be created in Unity by so call procedural mesh generation, where the 3D models are built in runtime from vertices and triangles connecting them. It requires, however, that the corresponding mesh generation algorithm is written by the developer.

6.2.4 Scripting system

The scripting system of Unity is based on Mono framework, which is an open-source substitute of the .NET framework from Microsoft. It allows using standard .NET methods and data types in the scripts, and it is also possible to include existing .NET libraries in the projects as built DLLs. In Unity projects, scripts are programmed by using either C# or JavaScript[1]. A single project can contain scripts written in both languages. Unity engine comes with the free MonoDevelop IDE for script development, but the engine can also be integrated to Microsoft Visual Studio to enable easy programming and runtime debugging of scripts.

The scripts that are used for controlling GameObjects are derived from *MonoBehaviour* class of Unity engine. When a *MonoBehaviour* script is attached to a GameObject, its certain methods are called automatically based on the situation of the object. E.g. after initialization of a GameObject the *Start* method of its script component is called. During program execution the status of GameObject is updated every frame by calling the *Update* method of the script. It is also possible to add multiple script components to the same GameObject.

Some of the most important methods provided by *MonoBehaviour* class are described in Table 1. By default, each new script contains empty implementations for *Start* and *Up-*

[1] Unity's version of JavaScript is also known as UnityScript, but the two terms are being mixed even in the official documentation. E.g. the scripting documentation states that UnityScript is a proprietary language modeled after JavaScript [65], while the official web page mentions only JavaScript. The term JavaScript is also used in Unity editor and in various scripting examples.

date, but methods can be added and removed as much as is necessary for defining the behaviour of the object.

Table 1: *Examples of the methods provided by MonoBehaviour class [66]*

Method name	Description
Awake	This method is called when the script component is being loaded.
Start	This method is called at the beginning of the execution, before calling any of the Update methods.
Update	This method is called every frame.
FixedUpdate	This method is called on fixed interval, regardless of the actual frame rate.
LateUpdate	This method is called every frame after all other Update methods have been called. It is useful e.g. for updating position of a camera that follows some other object at a fixed distance.
OnMouseDown	This method is called, when the user presses mouse button over a physical item belonging to the GameObject.
OnDestroy	This method is called when the GameObject or its script component is being destroyed.

The programming system of Unreal Engine is based on C++ and every object of the engine is essentially a C++ class. Program logic is added to objects by attaching *Script components* to actors. Adding a script component results in creation of a skeleton class, where the methods *Initialize* and *Tick* are used in a similar manner that *Start* and *Update* in MonoBehaviour scripts. [62] The scripting itself can be done in C++ or by using the visual Blueprint scripting language. The latter is based on graphical nodes, representing variables, functions and events. The desired functionality is created by connecting the nodes into visual networks.

An example of Blueprint scripting is given in Figure 17. The script calculates the distance between two vectors, given by *Point A* and *Point B,* converts the result into a string and prints it on the screen [67].

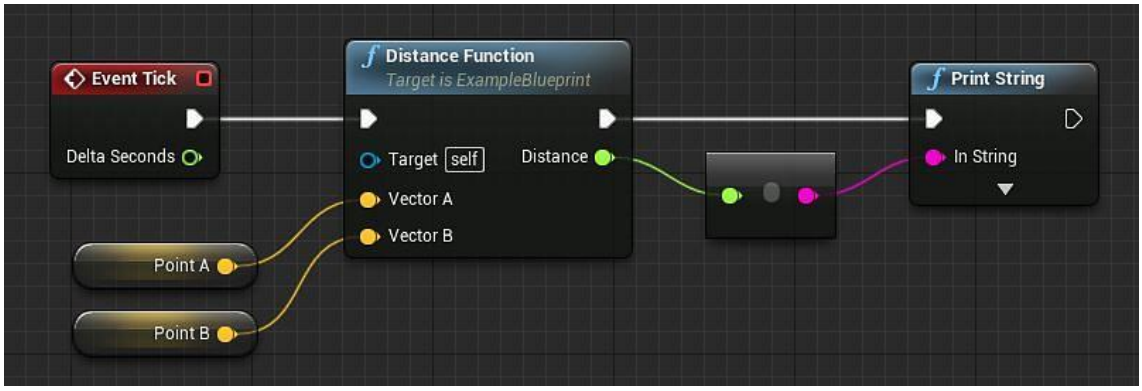

Figure 17: *Example of Blueprint visual scripting. The two functions that are used in this script are defined in separate networks. [67]*

The possibility to use visual scripting may be attractive for developers with no previous coding experience. However, relatively lot of nodes and connections are required for

defining an algorithm that would require only few lines of code, when using a high-level programing language. Modeling the entire behaviour of one CHE alone would lead to an awkwardly large network of functions. Blueprints are also known to run up to 10 times slower when comparing to C++ scripts with similar functionality [68; 69]. Thus, relying solely on Blueprints could lead to performance issues in terminals, where thousands of containers are added during initialization and hundreds of machine status updates are handled each second.

The runtime generation of terminal objects calls for a controller class, which handles the creation and initialization of objects based on the cached input data. Implementing such class might be difficult by using the Blueprint language. Furthermore, the C++ scripting system of UE is more complex than the C# syntax used in Unity, and there are not many detailed programming examples available for the former. This is a major disadvantage for UE.

6.2.5 Other features

Each of the engine providers is hosting a web store for distributing ready-made content, with items ranging from 3D models to sound effects and script packages. In April 2016 Unity Asset Store contained more than 25,000 items, of which more than 2,000 were available free of charge. The amount of items in Unreal Engine Marketplace was somewhat smaller and there was very little free content available.

Each of the engines allows using some external version control system directly from the editor. Even though the supported systems were different for the two engines, it was not considered an important factor for the decision process. Popular version control systems like Subversion and Git may be used through graphical client applications, which show all the changes in the local file structure at one glance. They can also be easily configured to ignore files that are not relevant for the version control. Thus, version control of a project can be easily achieved without having the version control system integrated to the engine itself.

6.3 Conclusion

While both engines have their advantages and disadvantages, Unity was chosen as the development platform for a variety of reasons: Firstly, it allows using of 3D models previously created in Kalmar. The existing models can be imported in the editor environment as such, and new models can be easily created by using Blender. Secondly, the engine has already been used in VR applications within Kalmar. Thus, the existing terminal models can be easily copied to a new project.

Unity was also very intuitive to use and easy to learn based on the training material. After spending two hours with a video tutorial, one was able to create prefabs and de-

velop behaviour scripts independently. There is also a vast amount of support material available in the Web, including plenty of tutorials, comprehensive programming reference, freely distributed code snippets, forum discussions and more. The object model of UE was much more difficult to grasp, and the training material for scripting seemed to concentrate on the Blueprint language. After spending several hours with documentation, no clear idea was formed of how to implement the proposed functionality with the engine.

Finally, there were well-documented .NET libraries available for wrapping the native Qt implementation of UniQ libraries. An example project was provided for establishing the communication with relatively few lines of C# code. Author's previous experience from .NET development in Visual Studio environment contributed to the choice of C# as the main programming language for the project.

Taking into account all these factors, a decision was made for Unity, as it was seen to allow fastest possible prototyping for the visualization tool.

7. SOLUTION DETAILS

The visualization tool was implemented with Unity (version 5.3.4) and Microsoft Visual Studio, using C# as the sole programming language. Since Unity is currently using relatively old version of Mono framework, some libraries in the dependency chain had to be rebuilt to target .NET version 3.5 or older. This caused some extra work in early phases of the implementation. However, once it was possible to initiate calls to UniQ libraries from Unity project, the application development was relatively straightforward.

Following sections describe the structure and technical details of the solution. External behaviour of the application is demonstrated in Appendix 1.

7.1 Structure of the application

Figure 18 illustrates the high-level architecture of the application. When comparing to the initial proposal in Figure 16, there are few differences. The back-end is split in three interconnected libraries: *FSBackendLib.Common*, *FSBackendLib* and *FSBackendLib.UniQ*. UniQ libraries are only referenced by the latter, i.e. the communication components are only visible within that module. *FSBackendLib.Common* is referenced by all other modules, allowing bi-directional communication between the Unity project and communication components.

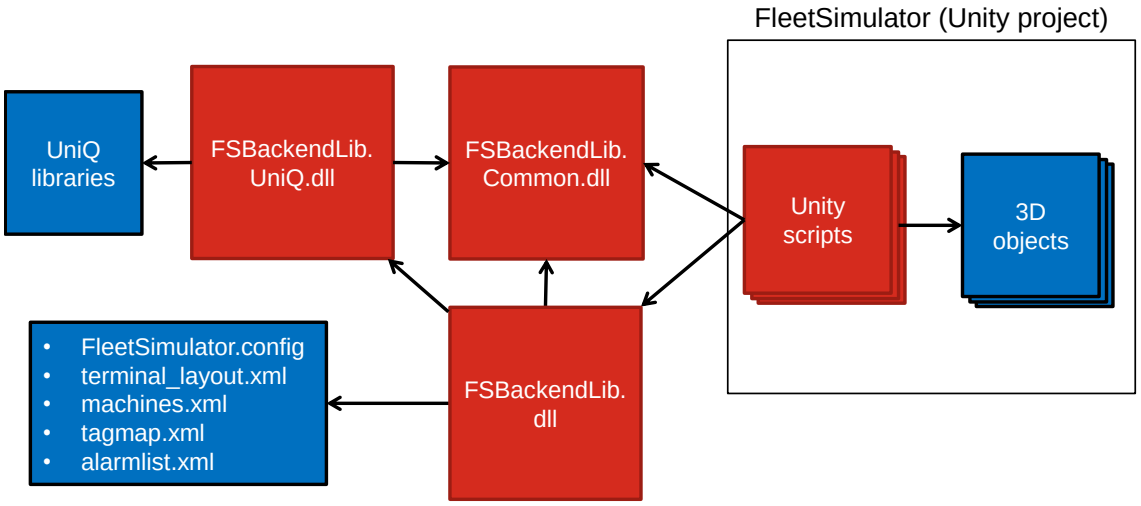

Figure 18: *High-level architecture of the application*

For giving the general input parameters to the application, a dedicated XML configuration file *FleetSimulator.config* was created. An example of the file syntax is given in Program 4. The file is used to define the location of other configuration files on the user

computer and the address of the Spread daemon, which is used for communication. The user is also able to choose between the dynamic terminal layout, which is parsed from *terminal_layout.xml*, and fixed layout options, which were created in Unity editor during design-time. The layout is chosen according to the *StaticLayoutType* parameter. The *SimulatorMode* parameter is used to define whether the application is used to visualize simulated terminal or actual systems.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- General configuration file for FleetSimulator -->
<FSConfiguration>
  <!-- Address of the Spread daemon -->
    <SpreadAddress>4803@127.0.0.1</SpreadAddress>
  <!-- Address of the parameter folder, containing tagmap.xml,
    alarmlist.xml, terminal_layout.xml and machines.xml. -->
    <ParameterFolderAddress>
    C:\FleetSimulator\parameters\example_terminal
    </ParameterFolderAddress>
  <!-- Source of the static terminal layout.
    Set "dynamic" to parse the layout from XML. -->
    <StaticLayoutType>dynamic</StaticLayoutType>
  <!-- Defines source of CHE status tags.
    Set true for simulated systems. -->
    <SimulatorMode>true</SimulatorMode>
</FSConfiguration>
```

Program 4: *Example of FleetSimulator.config*

Modules of the application are described in more detail in the following sections.

7.1.1 Structure of the back-end

Internal structure of *FSBackendLib.Common* is presented in Figure 19. Purpose of the module is to provide a common data model that is used by all parts of the application.

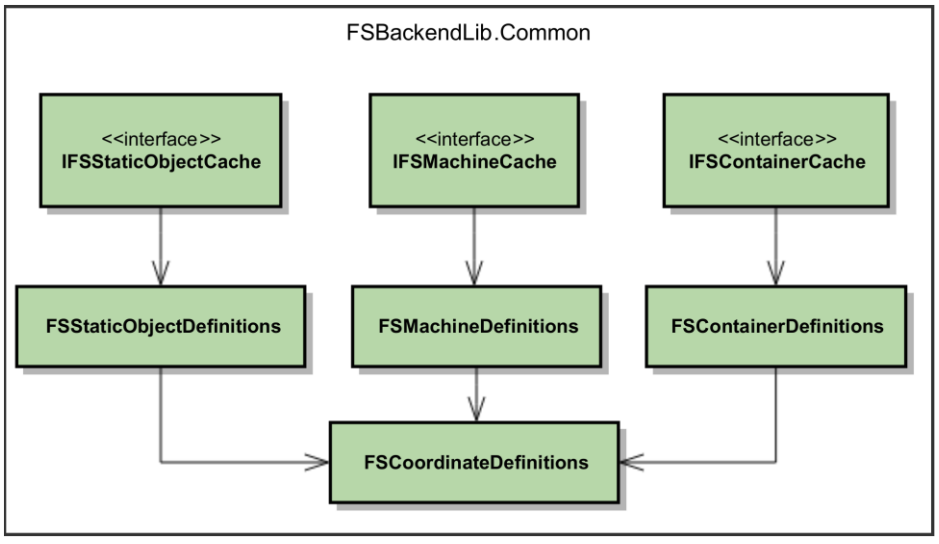

Figure 19: *Structure of FSBackendLib.Common*

In the application, objects of the terminal are divided in three logical groups, each of which has its own collection class called **cache.**

- **Static objects** represent the terminal objects, which are instantiated during startup, but have no active role in the simulation. These are parsed from *terminal_layout.xml* and include asphalt, water, rails, fences, buildings, etc.
- **Machines** include the physical machines of the terminal and the control systems, which are relevant for communication and visualization. These are parsed from *machines.xml* and used in both initialization and runtime communication.
- **Containers** include the containers of the terminal and the related concepts like stacks. These are obtained in runtime from the control systems.

In addition to these three groups, the back-end has its own **coordinate** types, allowing easy transformations from XML data to 2D space and from 2D space to the 3D coordinate system used in Unity. Further data structures are used for mapping an object's position to its parent system, as described earlier in Chapter 3.

The classes and data types defined in *FSBackendLib.Common* are visible to all other modules of the application, but the caches are only defined as interfaces. Their implementing classes are included in *FSBackendLib*, as illustrated in Figure 20.

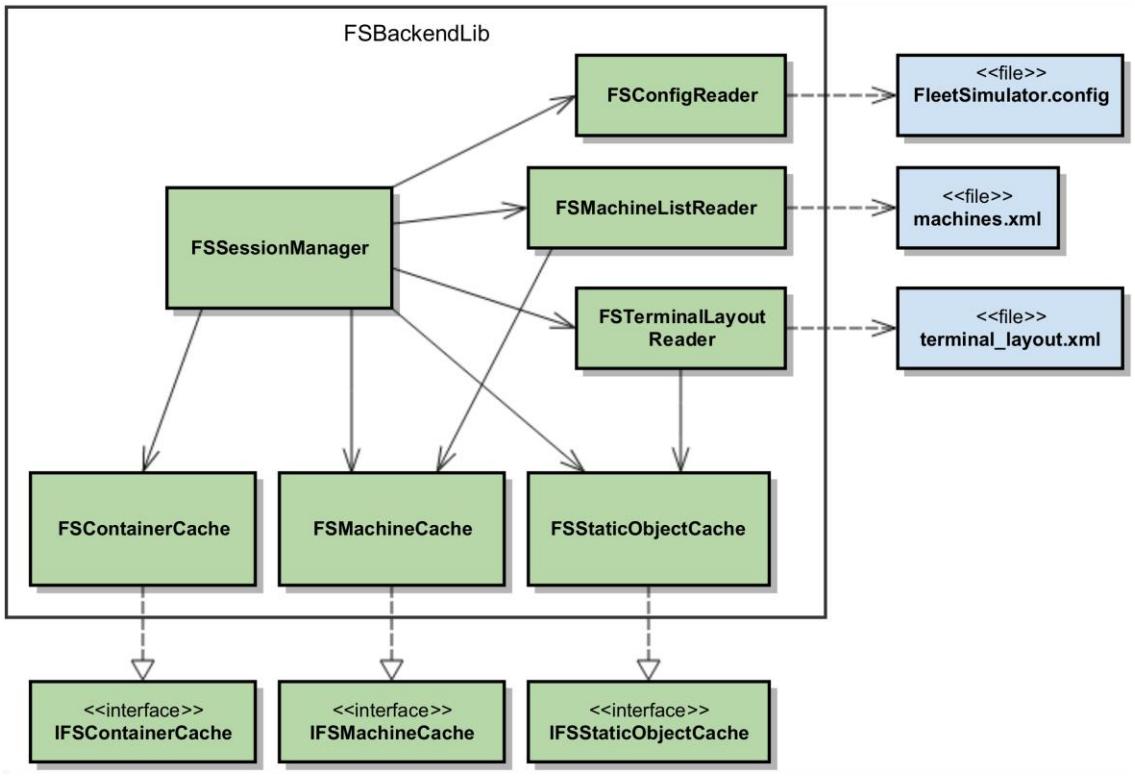

Figure 20: *Structure of FSBackendLib*

In addition to the cache implementations, *FSBackendLib* contains classes that perform the parsing of configuration files and insert the corresponding data objects to the caches.

Parsing of *tagmap.xml* and *alarmlist.xml* is performed in UniQ libraries, so these files are not handled separately in the back-end. Main class of the back-end is *FSSession-Manager*, which handles the input parameters from *FleetSimulator.config* and controls the program workflow by passing requests to other classes. It is also the only class, which is visible from outside of the library. This means that the cache implementations are hidden from the client.

It was initially planned that the cache implementations could be in some cases replaced with so called **stubber** classes to provide well-defined example data to be used with the Unity project. Writing and maintaining of the stub implementations proved to be more laborious than using actual process data from virtual test environments. Thus, the stubbers were eventually removed from the final solution.

Communication between the application and simulation environments is implemented in *FSBackendLib.UniQ*. Simplified structure of the module is presented in Figure 21. The library consists of a communication manager class and different types of tag listener classes. During the initialization *FSCommunicationManager* uses the machine cache to get a list of machines and instantiates one *FSTagListener* per each CHE or control system. Tag listener classes contain the logic for ordering and parsing tags and for passing the updated data to the respective objects in caches.

An example of the interaction between classes is given later in section 7.1.3.

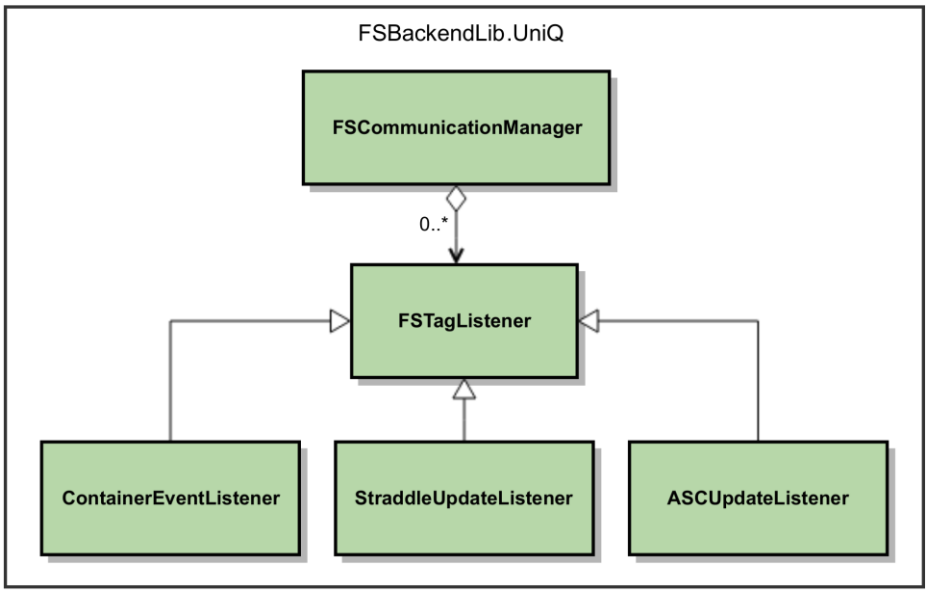

Figure 21: *Simplified structure of FSBackendLib.UniQ. External communication components by Kalmar and other parties are excluded from the diagram.*

In order to allow easy testing of back-end components, a simple test application was written to be used with Visual Studio debugger. The test application acts as a client for the back-end libraries and uses their public methods like the Unity project would do. Instead of visualizing the items, the application provides a console output for the log

messages of *FSBackendLib.UniQ* as shown in Figure 22. The greatest benefit from the test application is the possibility to run the back-end code step-by-step, see the status of all objects and insert breakpoints into code. Since the Unity project uses back-end components as built DLLs, it is not possible to do detailed analysis of the code behaviour when using the libraries in Unity environment.

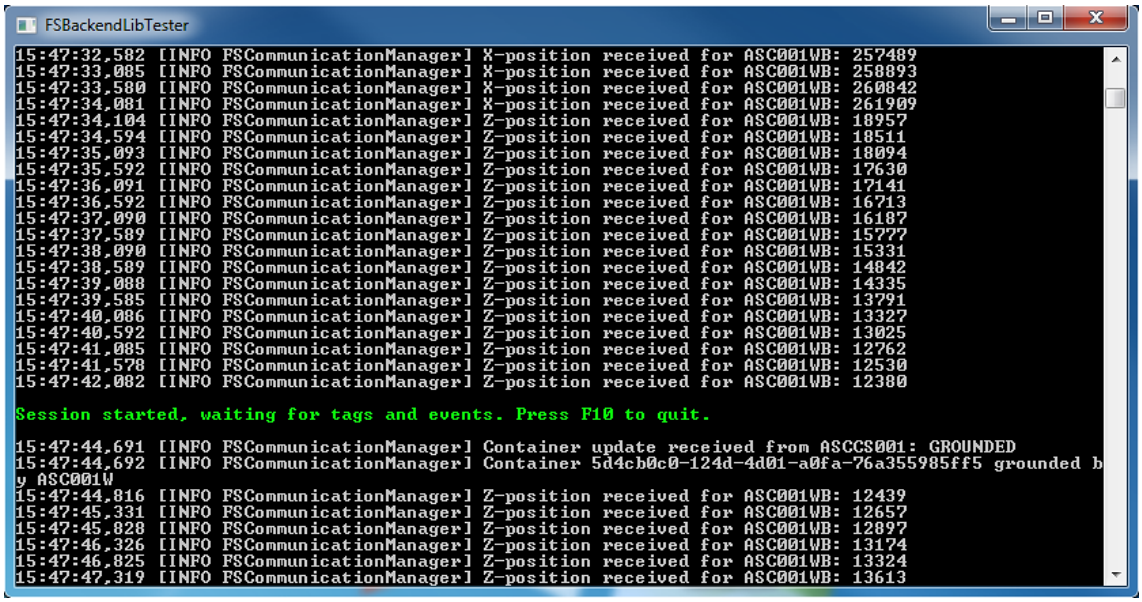

Figure 22: *UniQ traffic logging from the test application*

7.1.2 Structure of the Unity project

The Unity project consists of five scenes. One of the scenes acts as a loading view, which is shown to the user during startup. The other four scenes represent the different layout options, one of which is loaded according to the input parameter value from *FleetSimulator.config.* Structure of the *dynamic layout* scene is presented in Figure 23. The scene uses the cached data from *terminal_layout.xml* to build the static terminal layout during runtime.

SceneManager is the main organizational unit of the scene and it uses the *SceneController* script to call the services of the back-end. The three additional *Handler*-objects act as organization units for static objects, machines and containers respectively. Their controller scripts make calls to the respective caches and instantiate the visible objects according to the cached data. Each CHE object has its own controller script, which is polling the status of the corresponding machine object in the cache. Containers have no scripts attached to them. Thus, when not manipulated by *ContainerController* or some other object, the containers act like static objects, lying passively in the scene.

The other three scenes have almost similar structure, with the exception that static objects have been defined in design time and there is no *StaticObjectHandler* or *StaticObjectController*. These scenes may also be used without *terminal_layout.xml* as the static

object cache is not used at all. Figure 24 illustrates a static layout scene based on the Kalmar test site in Tampere.

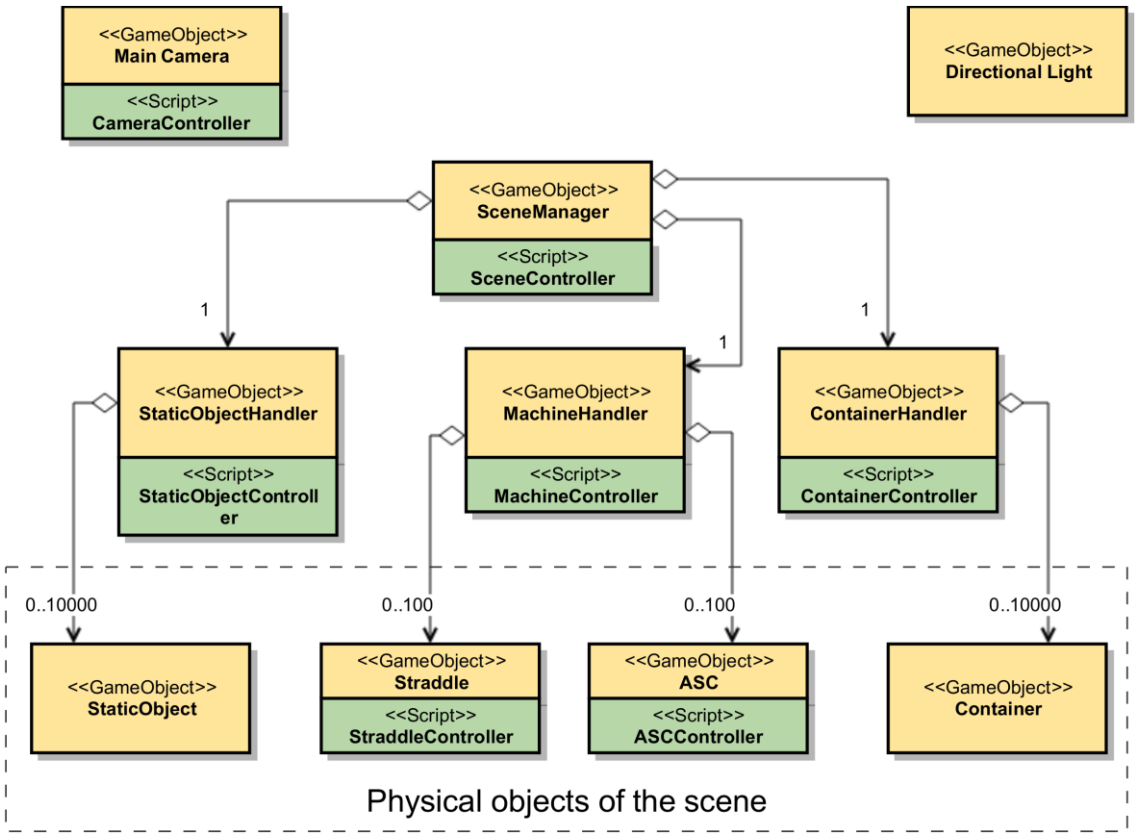

Figure 23: *Structure of the dynamically built Unity scene*

Figure 24: *Prebuilt static layout scene based on the Kalmar test site in Rusko, Tampere*

7.1.3 Communication between modules

Figure 25 gives an example of the communication between classes of the application, performing initialization and data visualization for a single straddle carrier object. Step-by-step description of the process is given below.

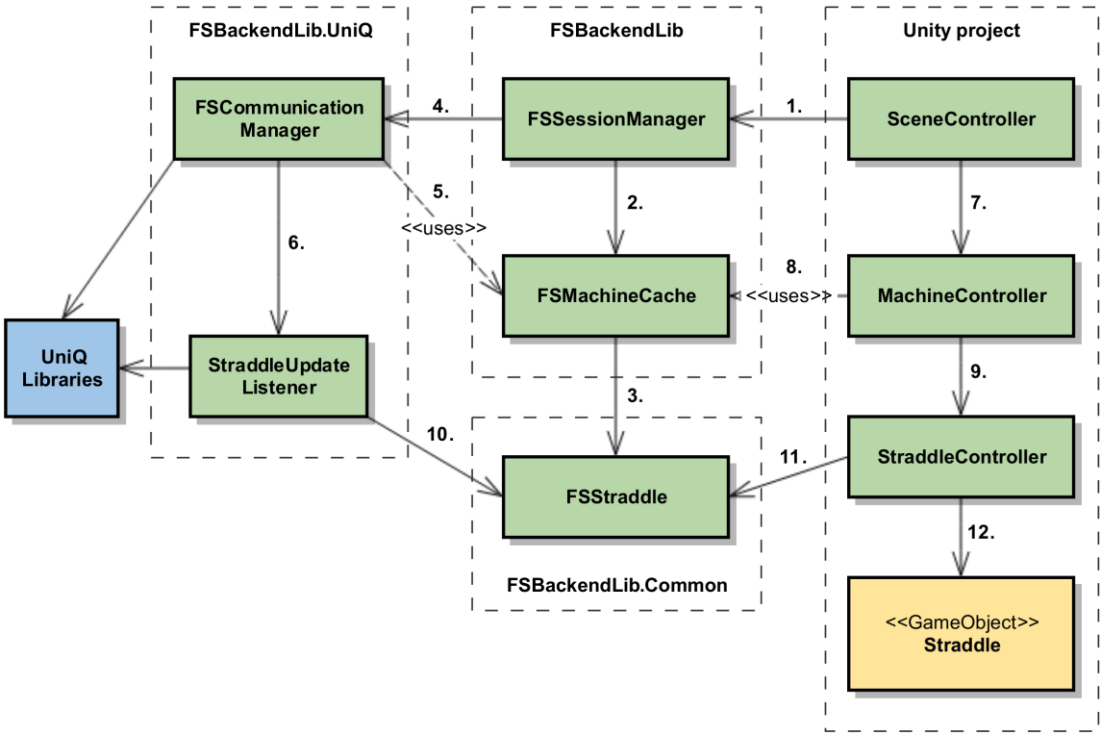

Figure 25: *Example of the initialization and usage of a straddle carrier object. Some steps are omitted from the diagram to maintain readability.*

Steps 1 – 9 are performed once during startup and initialization.

1. *SceneController* calls *FSSessionManager*, asking for a reference to the machine cache.
2. *FSSessionManager* initializes an *FSMachineCache* object.
3. An object of type *FSStraddle* is created and added to the cache. Effectively this is achieved by calling *FSMachineListReader*, which parses *machines.xml* and populates the cache with corresponding machine objects.
4. *FSSessionManager* calls the initialization method of *FSCommunicationManager*.
5. *FSCommunicationManager* calls the machine cache through interface and requests a list of machines.
6. A *StraddleUpdateListener* is created with a reference to the cached *FSStraddle*.
7. Reference to the machine cache is returned to *SceneController*, which passes it to the initialization method of *MachineController*.
8. *MachineController* uses the machine cache through interface and requests a list of physical machines.

9. *MachineController* instantiates a straddle carrier object and initializes its controller script with a reference to the cached *FSStraddle*.

After the initialization, steps 10 – 12 are repeated as long as the application is running.

10. *StraddleUpdateListener* converts the arriving tag values into updates to the cached *FSStraddle*.
11. – 12. *StraddleController* reads the cached values on each frame from *FSStraddle* and updates the state of the visible straddle model according to the data.

StraddleUpdateListener consists of multiple tag receivers, each of which has a delegate to the respective handler method. When a tag is received, its handler method is called automatically in a separate worker thread, which is assigned by UniQ libraries. Thus, the receiving and updating of data is fully parallel to the visualization.

7.1.4 Reasoning for the architecture

The implemented architecture satisfies the previously stated goals of tailored data model and limiting the responsibilities of Unity scripts. It represents a layered structure, with clear responsibilities on visualization layer, data storage layer and communication layer.

The application is following a rough MVC (Model-View-Controller) kind of pattern, where the *model* part comprises the whole back-end with the related configuration files and external systems. Unity scripts and user controls represent the *controller* part, which uses the 3D models and visible GUI components to create the *view*. Structure of the back-end allows creating multiple different *views* for the same *model*, with one *view* being the previously described test application.

The solution uses dependency injection to decouple the Unity scripts from the communication components. *FSSessionManager* acts as a factory, creating and populating the cache objects for the client. *FSCommunicationManager* acts as an injector, establishing the communication and initiating updates to the caches. The Unity project knows only about the session manager and the common data model and is not interested of how the cached data is being acquired.

Dependencies to any third-party libraries are encapsulated within *FSBackendLib.UniQ*, while the other modules of the back-end are written solely on standard .NET libraries. The back-end is also independent of any data structures defined in Unity engine, so it could theoretically be used with other visualization engines as such or through a suitable wrapper interface.

7.2 Handling of 3D objects

The application makes extensive use of Unity's prefab system in the creation and handling of 3D objects. E.g. a prefab for a straddle carrier contains a real-size 3D model of the machine, which is by default positioned in the origin of the terminal, facing positive X-direction. The prefab contains also the controller script, which is being copied along with prefab instances. Thus, each visible straddle object is controlled by its own instance of the *StraddleController* class.

Prefabs can be used as parameters for other scripts, as is illustrated in Figure 26. The prefab for automated shuttle is given as parameter for *MachineController* script, which is attached to *MachineHandler* object of the scene.

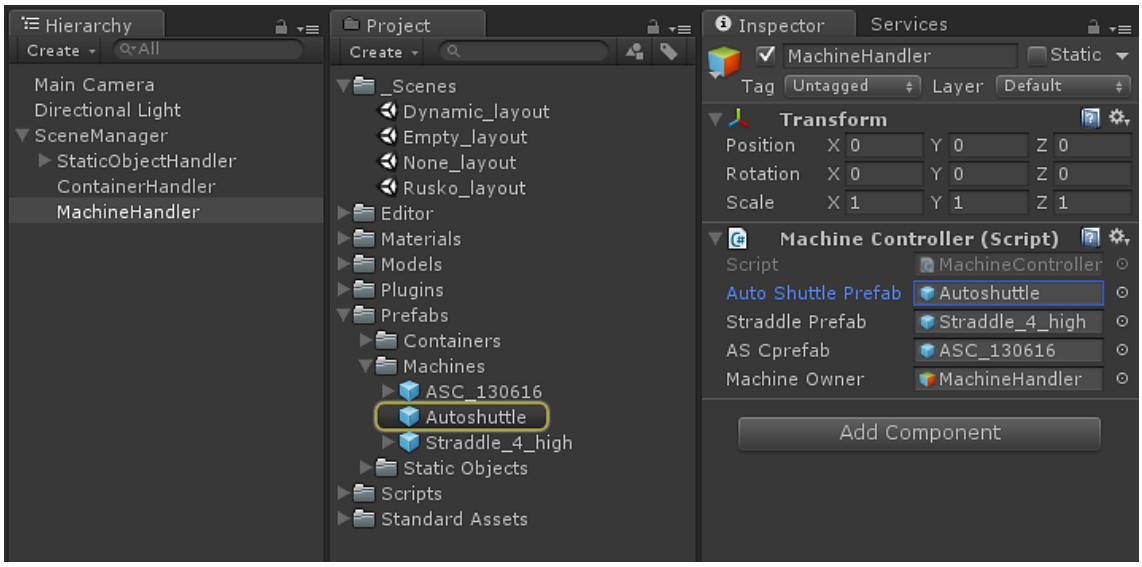

Figure 26: *Example of using a prefab as a parameter for Unity script*

Prefabs, like any other GameObjects can be used as variables in code. Program 5 contains the Unity commands for instantiating a prefab for an ASC and initializing its controller script based on the cached data. Position of the object is defined as *Vector3* and its rotation is set by using the *Quaternion* data structure.

```
FS_ASC cachedASC;         // Cached data of the ASC
GameObject ASCPrefab;     // Template for an ASC object
 // Get the initial position and heading for the machine
var position = cachedASC.GetPosition();
Vector3 positionVector = new Vector3(position.x, position.y, position.z);
float rotationAngle = cachedASC.GetRotation();
Quaternion rotation = Quaternion.AngleAxis(rotationAngle, Vector3.up);
// Create an ASC object according to the data
GameObject ASCObject =
      Instantiate(ASCPrefab, positionVector, rotation) as GameObject;
ASCObject.name = cachedASC.GetName();
// Initialize the controller script of the newly created ASC object
ASCController controller = ASCObject.GetComponent<ASCController>();
controller.Initialize(cachedASC);
```

Program 5: *Unity commands for creating an ASC object based on the cached data*

The 3D model, which is used in ASC prefab, is presented in Figure 27. The *Hierarchy* panel on the right shows the internal structure of the model, which forms a tree of parent-child relationships. E.g. the *Spreader* object is a grandchild of *Trolley*. Thus, whenever the trolley is moved, the spreader moves along with it. When the spreader is moved separately, it does not affect the position of trolley or any other components higher in the hierarchy.

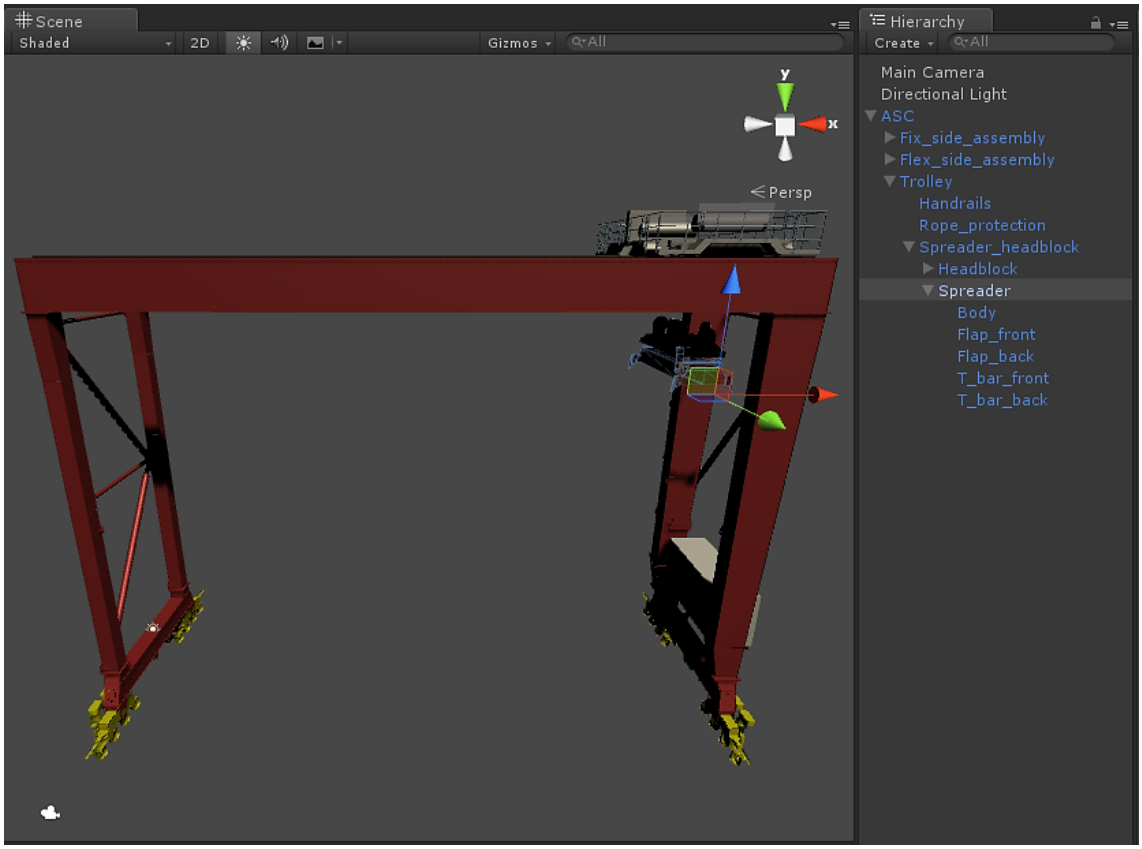

Figure 27: *ASC model and its subassemblies*

Each subcomponent of a model can be considered as a separate GameObject. Program 6 demonstrates the Unity commands for getting the position and orientation data for the ASC object and its subassemblies.

```
// Find a CHE instance named "ASC"
GameObject ASC = GameObject.Find("ASC");
// Get the position of "ASC" in terminal coordinates
Vector3 ASCPosition = ASC.transform.position;
// Get the rotation of "ASC" around vertical axis
float ASCHeading = ASC.transform.eulerAngles.y;
// Find a subassembly named "Trolley"
GameObject trolley = ASC.Find("Trolley");
// Find a further subassembly named "Spreader"
GameObject spreader = trolley.Find("Spreader_headblock/Spreader");
// Get the local and terminal coordinates for the spreader
Vector3 spreaderGlobalPosition = spreader.transform.position;
Vector3 spreaderLocalPosition = spreader.transform.localPosition;
```

Program 6: *Unity commands for accessing the ASC model and its subassemblies*

In the end of Program 6, two different positions were obtained for the *Spreader*: **Global position** describes the position of the object in common scene coordinates, while its **local positon** describes its position in relation to its parent. The **local transform** of the *Spreader* object can be seen in Figure 27. The coordinate axes of the transform are different from those of the scene.

The possibility to use local coordinates makes it easy to apply trolley and hoist positions to the ASC objects: The crane's trolley position is measured as its offset from a fixed point on the gantry. Position of the spreader (or hoist height) is measured as the vertical distance between the spreader and the ground. When this information is applied to the 3D model, it is sufficient to change an object's position along one axis in its local coordinate space. The model hierarchy guarantees that all relevant components are moved along with the manipulated object.

Global position of the spreader is used, when attaching containers to the spreader. The container handling also takes advantage of the parent-child relationships, as is demonstrated in Program 7. For the time between picking and grounding, the container object is made a child of the spreader. This makes the container move along with the crane until the time it is grounded, and no separate code is required for animating its movement. Structure of the ASC object after applying the code is illustrated in Figure 28.

```
1  // Find a container object named "SampleContainer"
2  GameObject container = GameObject.Find("SampleContainer");
3  // Position the container into the spreader
4  container.transform.position = spreader.transform.position;
5  // Make the container a child object of the spreader
6  container.transform.parent = spreader.transform;
```

Program 7: *Unity commands for assigning a container to the spreader*

In *MonoBehaviour* scripts, the object owning the script can also be manipulated without declaring the GameObject separately in the code. E.g. in the case of Program 6, the expression `transform.position` would be equal to `ASC.transform.position`, given that the script itself is a component of *ASC*. In the code examples of this thesis, the manipulated GameObjects are always declared for clarity reasons.

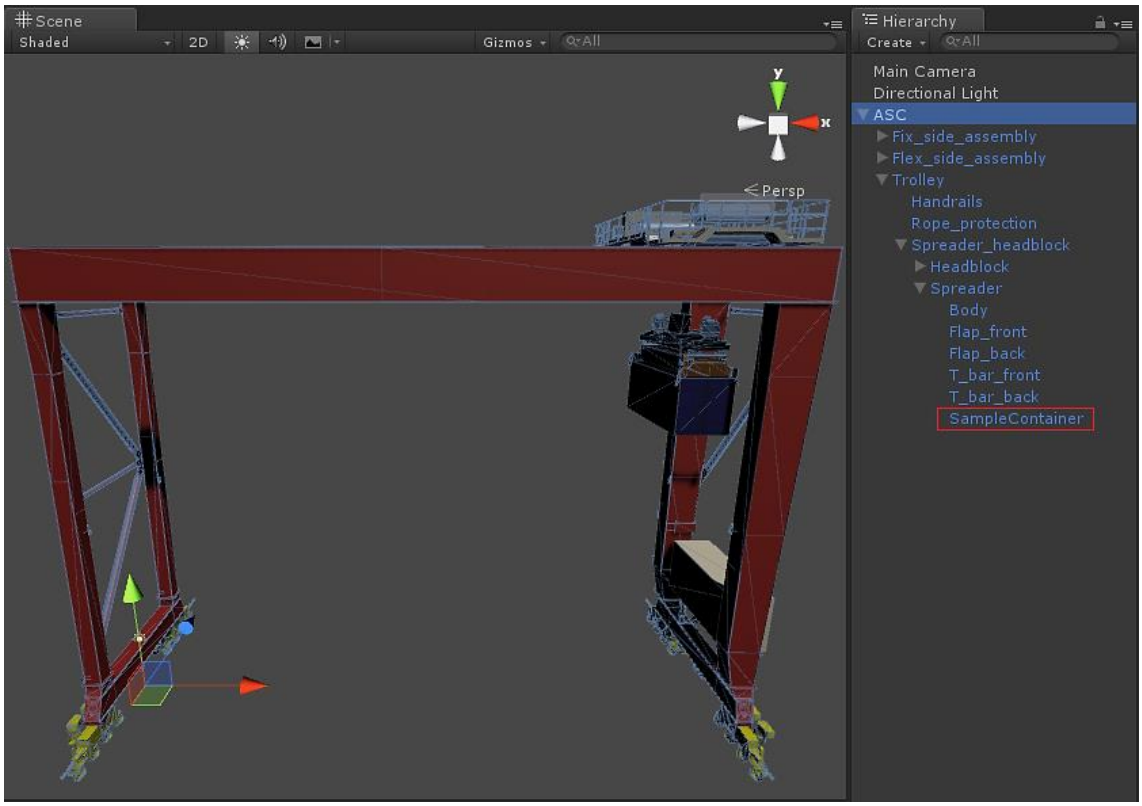

Figure 28: *ASC model with the picked container*

7.2.1 Comments on the used methods

The previous, simplified code examples make extensive use *GameObject.Find*, which searches for a GameObject according to its name. Using the method is considered as bad practice by some developers. One argument against use of the method is that it needs to iterate through all GameObjects of the scene to find the desired object [70]. This may lead to performance issues, when the method is used frequently [71].

In *FleetSimulator* code the machine subcomponents are stored in member variables of the controller scripts during the initialization. Thus, the *Find* method is only used by CHE controller scripts during startup. Container finding is performed every time when information about pick or ground event is received from the back-end. This may happen up to few times per second in larger terminals, but using *GameObject.Find* does not seem to cause any performance loss, when applied only sporadically. If such problem would arise, a separate list with references to container objects could be used to completely avoid using of the method.

Another argument against *GameObject.Find* is that it creates hidden dependencies by making assumptions of objects' names [70]. In *FleetSimulator* application the names are typically obtained from the back-end during runtime, or they are properties of the 3D model, which the controller script is supposed to manipulate. Thus, whenever a model's naming is modified, its controller script needs to be updated as well.

7.3 Handling of container instances

Since there is no direct link between TOS and the UniQ platform, container events are sent to UI by the control systems. Container events are communicated by using update messages, which consist of basic data of the container, optional location information and a status field. Possible status updates for a container include *ADDED*, *PICKED*, *GROUNDED* and *DELETED*, among few others.

Like was described in Chapter 3, each CS has its own interpretation of the containers in its operational area. These "container maps" overlap in the interchange areas of ASC blocks. As the container moves through the terminal, it will eventually be registered by various control systems.

Figure 29 illustrates a situation, where a container has three simultaneous representations in the cache. The container was initially located in an ASC block controlled by *ASCCS002*. It was then picked by a straddle and moved to transfer area of another ASC block. From viewpoint of *ASCCS002* the container is *DELETED*. *HTCS* sees it as *GROUNDED* and knows the exact X-Y-position where the container was left. The new position is shared with *ASCCS001*, which sees the container as *ADDED*. From the update by *ASCCS001*, only the logical slot position of the container is known.

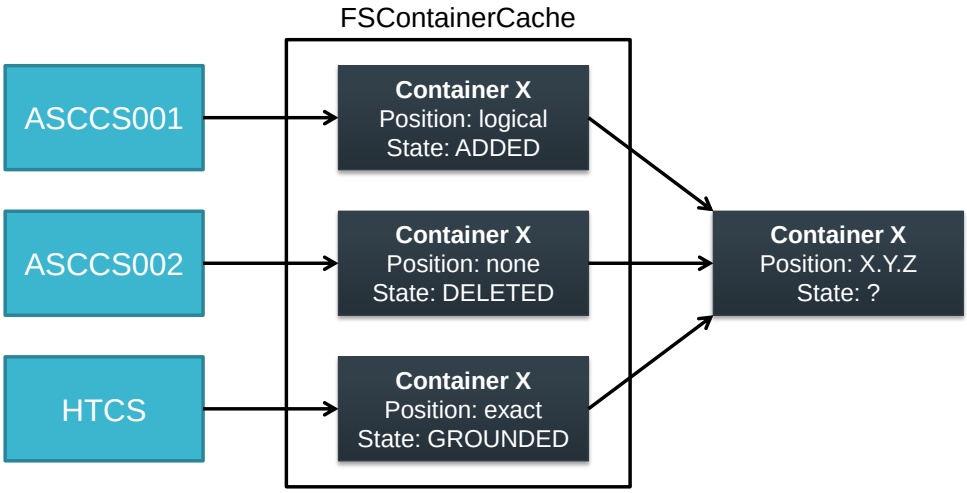

Figure 29: *Multiple cached instances of the same container, with varying statuses*

In the 3D world there can be only one physical instance of each container and every visualized container must have a well-defined 3D position. *FSContainerCache* contains the logic for solving container's actual state from multiple ambiguous container updates. The logical stack positions are converted to exact positions based on the stack information, which is received from the CS during startup. Vertical position of each container is calculated based on its tier number and the height information of the containers lying lower in the same stack. Internal structure of *FSContainerCache* is illustrated in Figure 30.

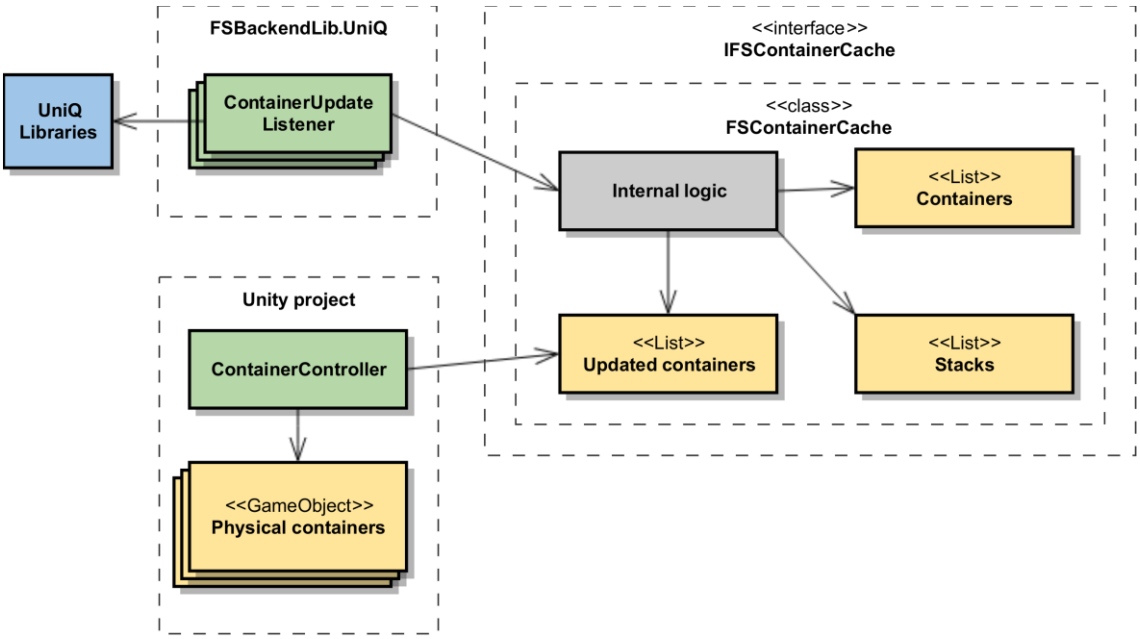

Figure 30: *Structure and working principle of FSContainerCache*

The cache contains separate lists for container and stack information. These lists allow storing of multiple container or stack instances with same name, as long as they are received from different control systems. The UniQ part of the application contains one *ContainerUpdateListener* per each CS. Internal logic of the cache is used to solve the actual states of the containers and to store the resulting updates in the *Updated containers* list. The information is then handed over to the *ContainerController* script, which applies the changes to the physical container objects of the scene.

All the mentioned list structures in *FSContainerCache* are protected with mutual exclusion mechanisms to prevent concurrent read and write operations, or writing attempts by multiple threads. Exclusive access to the list ensures that the list remains unmodified by other threads for the duration of each operation. This is especially important during initialization, when there may be more than 10 *ContainerUpdateListeners* storing their updates in the cache simultaneously. The *Updated containers* list is also acting like a buffer, which is emptied after each update request from *ContainerController*.

Due to the large amount of calculation that takes place during the parsing of container and stack information, some delay is added between operations to prevent excessive CPU usage. The amount of spawned 3D containers per frame is also limited to keep the UI navigable during initialization.

In the case of Figure 29 the visualized position would be the average of the positions given by *HTCS* and *ASCCS001*. If one of the positions could not be resolved, e.g. due to missing stack information, the container would be visualized according to the other instance.

7.4 Movement algorithm for machines

Unity provides built-in mechanisms for creating animations for objects. These mechanisms involve programming the magnitude and timing of movements into pre-defined curves, which can be looped to emulate continuous movement. In the case of simulated CHE positions, both the magnitude of movement and its speed are a priori unknown. Thus, implementing the movement with animations would require extensive parametrization and/or modification of animations during runtime. Animations could however be used to visualize events that are not based on constant change of tag values, such as opening of a gate or arrival of a road truck.

Program 8 describes an algorithm for converting cached position values into movement of 3D object by manipulating its position directly. Generic *var* variable type is used to represent data types, since the actual cached types depend on the target object. In *FleetSimulator*, algorithms like the one presented are used to handle 3D vectors, linear float values and rotation angles.

```
01  var cachedMachine;  // Cached data of the machine
02  GameObject Machine; // 3D-representation of the machine
03  float animationInterval;  // Time span for the animation
04  var previousPosition;
05  var targetPosition;
06  // This method is called once per frame
07  void UpdatePosition()
08  {
09     // Test if the target value has changed
10     var cachedPosition = cachedMachine.GetPosition();
11     if (cachedPosition != targetPosition)
12     {
13        previousPosition = Machine.transform.position;
14        targetPosition = cachedPosition;
15     }
16     // Animate the movement frame by frame
17     var currentPosition = Machine.transform.position;
18     if (currentPosition != targetPosition)
19     {
20        float timeStep = Time.deltaTime / animationInterval;
21        var movementStep = timeStep * (targetPosition - previousPosition);
22        var diffPosition = targetPosition - currentPosition;
23        if (movementStep <= diffPosition)
24        {
25           Machine.transform.position += movementStep;
26        }
27        else
28        {
29           // The target is closer than one step away
30           Machine.transform.position = targetPosition;
31        }
32     }
33  }
```

Program 8: *Generic movement algorithm for a machine*

The *UpdatePosition* method is called every frame by the machine's controller script. The first *if* statement is used check if the cached value has changed since the previous

frame. The difference between *previousPosition* and the tagged *targetPosition* is then animated over the time span defined in *animationInterval*. The method *Time.deltaTime* returns the amount of time that has passed since the last frame. It is used to calculate the amount of movement that is required for the current frame, thus keeping the speed of movement constant regardless of the application frame rate.

If the target has not been reached before a new cached value is received, a new *previousPosition* is calculated based on the latest achieved position. If the cached value does not change, the object will eventually reach its *targetPosition* and both of the higher level *if* statements will evaluate as false.

Every machine type in *FleetSimulator* has multiple movement algorithms, which are evaluated one after another. For an ASC, the position is updated separately for gantry, trolley, hoist and spreader length. For shuttles and straddles, updates are made in position, orientation, hoist height and spreader length. Algorithms are called in the *Update* method of the machine's controller script, and the changes are visualized after all *Update* methods have been completed.

7.4.1 Calculation of the animation interval

When tags are being requested from the simulation environment, the requested time interval between consecutive values is defined as T_{tag_order}, which by default has a value of 500 ms. In order to calculate the actual data update interval T_{update} as perceived by the movement algorithm, some **delay variables** need to be introduced:

- Communication delay $t_{communication}$ represents the difference between T_{tag_order} and the actual time from the previous received tag value to the next one. Its value varies according to the workload of the sending system and the network delay.
- Writing delay t_{write} is the time that is used to parse the received tag value and store the corresponding data in the cache. Parsing of position tags is usually very fast operation, but if the system is loaded by multiple concurrent write operations, t_{write} may be up to some milliseconds.
- Reading delay t_{read} is the time that passes before the new cached value is read by the movement algorithm. Unity application scans through all the update methods of the scene before rendering each frame, and its frame rate may be around 10 FPS (frames per second) at lowest. Thus, t_{read} can be anything between 0 – 100 ms.

The actual time between two values registered by the Unity application is now given by

$$T_{update} = T_{tag\_order} + \Delta t_{communication} + \Delta t_{write} + \Delta t_{read}. \quad (2)$$

In the current solution, the animation interval $T_{animation}$ is defined as

$$T_{animation} = T_{tag\_order} + D_{animation}, \quad (3)$$

where $D_{animation}$ is a constant value. In the ideal case $T_{animation}$ should be about equal to T_{update} to keep the animation synchronized with the tagged values. If $T_{animation}$ is much smaller than T_{update}, the machine will reach its target position and stop before a new target position is received, which makes the animation look bumpy. If $T_{animation}$ is much larger than T_{update}, the machine will start moving slowly compared to the tag values and eventually gain speed while trying to catch up. Practice has shown that setting $D_{animation}$ to 200 ms provides sufficiently smooth and timely animation in most situations.

7.4.2 Possible improvements

One option for making the animation follow the tag information precisely would be to store the values in the cache together with their time information. This would require a timestamp to be stored with each tag value and a new animation interval to be calculated for each updated position. This would add some complexity and computational workload to the application.

In some situations it would also be possible to get the speed information for machines in separate tags. Using these values in animation would however require a far more complicated algorithm, as the time difference between receiving a position tag and a speed tag could be anything from zero to T_{update}. Using the speed information to predict positions would also mean that the position information from the simulation environment would be replaced with a newly calculated value. As the application is only supposed to visualize information that is produced in the simulation environment, the current approach of animating the movement based on the latest known position is sufficient.

7.4.3 Handling of rotations

Calculating rotations with the previously discussed algorithm requires special attention due to the nature of heading angles: values 0° and 360° result as same orientation, and there may even be jumps from negative values to positive, and vice-versa.

In order to perform reliable calculation, all incoming values from the back-end and 3D models must be first converted to 0° – 360° range. Even then, there is a risk of erroneous behaviour, when passing the 360° limit. The situation is demonstrated in Table 2. Each row of the table represents an update due to a new tag value.

Table 2: *Example behaviour of rotation angles, when passing the 360° limit*

i	Previous angle	Target angle	Difference
1	335°	345°	10°
2	344°	356°	12°
3	355°	5°	-350°
4	5°	14°	9°

During the first two updates the model would be turning steadily towards the 360° orientation. The third update would cause the object to suddenly spin almost a full circle in the opposite direction and then continue the movement as it did before the third update. Similar situation would arise, when approaching the 0° orientation from above.

To prevent this type of errors, the value of *Previous angle* needs to be corrected, when a large change in target angle is observed. The *Current angle* value, which is obtained every frame from the 3D object, needs to be corrected as well. After applying the corrections the situation would look like in Table 3.

Table 3: *Example behaviour of rotation angles after applying the correction*

i	Previous angle	Target angle	Difference
1	335°	345°	10°
2	344°	356°	12°
3	~~355°~~ -5°	5°	10°
4	5°	14°	9°

8. TESTING AND EVALUATION

Purpose of this chapter is to describe the testing procedures and analyze the functionality of the application against the requirements that were presented earlier in Chapter 5. Section 8.1 covers the performance tests that were used to examine the behaviour of application, when working with large amounts of terminal objects. Further development ideas and requirements, which were not implemented during the thesis project, are discussed in section 8.2.

8.1 Performance testing

The application was primarily tested with virtual terminal environments, which include control systems and other components of a real TLS. Fleetview was used to create container events and jobs for the simulated CHE. Objects of the terminal were then visualized in *FleetSimulator*, allowing easy comparison between the two GUI applications.

For testing the application against requirement R 22 from Chapter 5, a virtual example terminal was created based on a real customer site. The environment included 10 functional ASC blocks with 19 cranes plus 28 straddle carriers, totaling in 47 CHE and several control systems. Separate test tools were used to add 10,000 containers to the environment.

In order to maximize the movement of machines, a small utility application named *TagBomber* was used to send simulated tag values to the environment. The straddles were made to drive on diagonal paths, while constantly changing their orientation and hoist position. ASCs were made to drive back and forth in their block, while constantly changing their trolley position, hoist position and spreader length. Various tag sending intervals were used for different machines, resulting in approximately 600 tags per second for the whole terminal. The *FleetSimulator* project was run in Unity editor on a separate computer. Technical specifications of the test computer are listed in Table 4. Figure 31 shows a view of the simulated terminal during the test.

Table 4: *Technical specifications of the test computer*

CPU	Intel Xeon E5-2620 v3 @ 2.40 GHZ
Memory	24.0 GB RAM
GPU	NVIDIA GeForce GTX 960
Operating system	Windows 7 Professional with Service Pack 1
Unity version	Unity 5.3.4f1

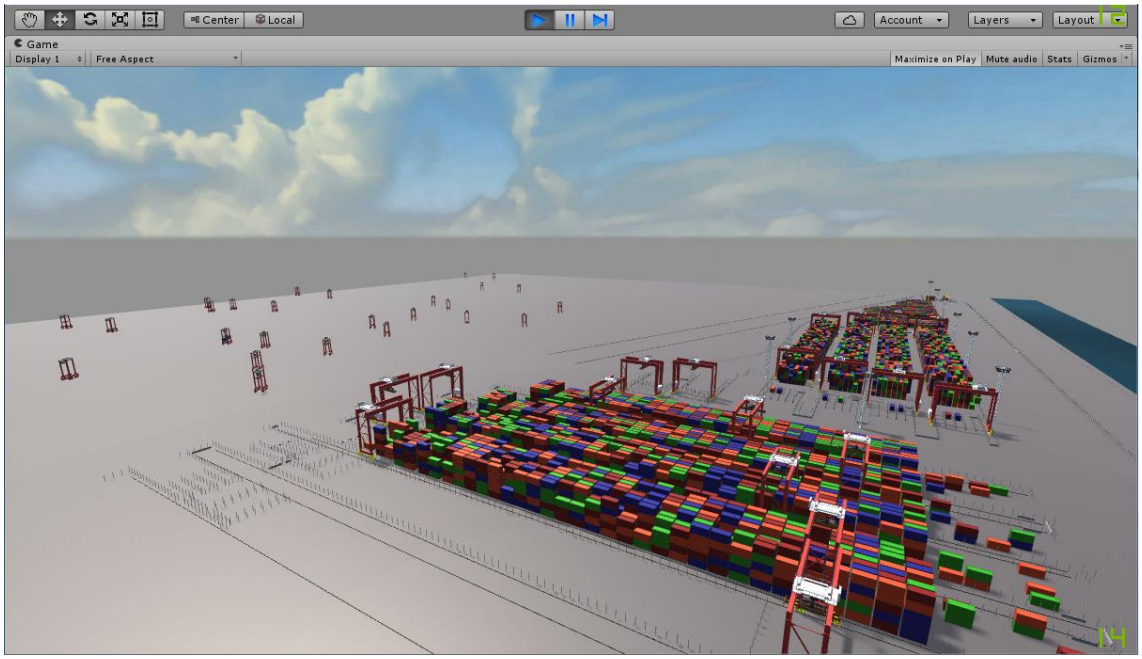

Figure 31: *A view of the simulated example terminal during the performance test*

The profiler window of Unity was used to analyze the application's performance during the test. An example of profiler's output is shown in Figure 32. The light green area of the CPU curve represents the amount of computation time that was used for rendering the graphical view. Execution of scripts, which is shown in cyan, represents less than 2% of the total computation time on the given frame. This number is possibly not including all the data processing that takes place in the back-end (see 8.1.2).

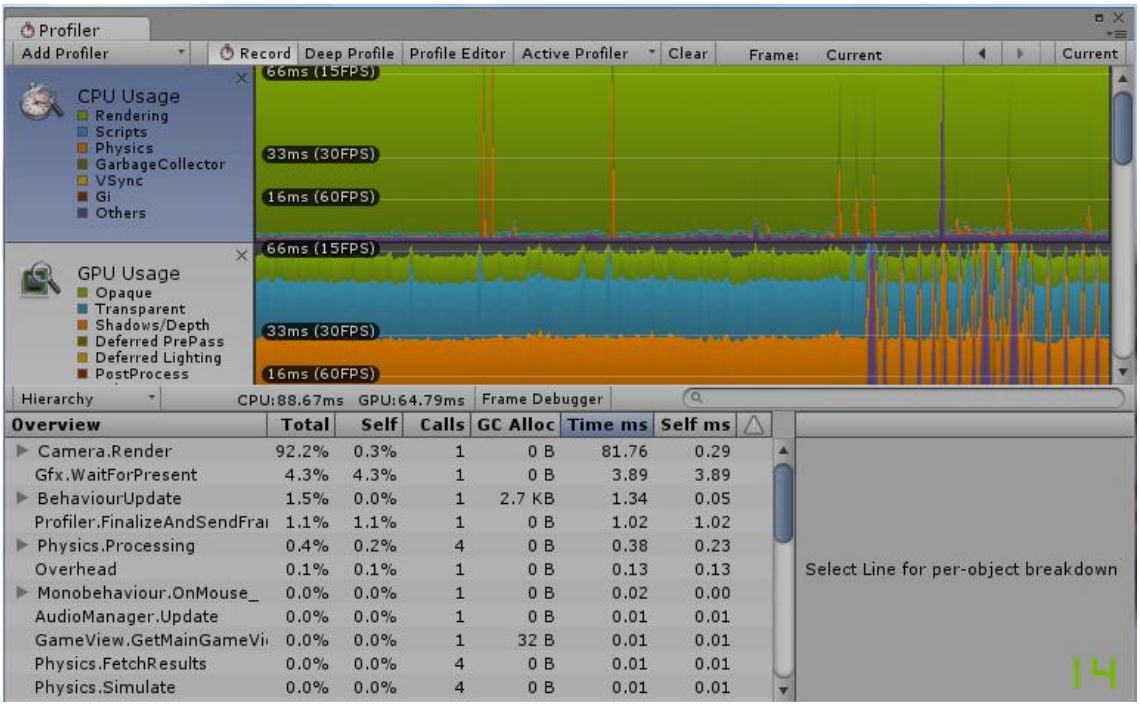

Figure 32: *Unity profiler output during the performance test*

Profiling the view from Figure 31 resulted in frame rates slightly above 10 FPS. When the camera was turned up towards the sky, the frame rate soared up to around 60 FPS, where it was kept by synchronization delay. Similar numbers were observed when all the machines were stopped. Thus, the amount of movement was not causing notable changes in application behaviour. However, in order to provide a smooth user experience, the frame rate should be kept at least 30 FPS in all situations. Techniques for optimizing the graphical performance are discussed in the following section.

8.1.1 Techniques for improving the frame rate

When analyzing the view of Figure 31 with tools provided by Unity, it was seen to consist of more than 20,000 meshes and have over 10 million triangles rendered on each frame.

Following techniques were found useful for optimizing the graphical performance:

- Limiting the visibility range of shadows or disabling them completely. Disabling of the shadows during the test situation dropped the amount of rendered triangles to around 4 million and raised the frame rate immediately to about 20 FPS.
- Limiting the visibility range of small objects, such as fence posts.
- Using as few individual meshes and materials as possible for modeling the objects. Each mesh should also contain as few polygons as is necessary for the given detail level. [72]
- Using simplified models for objects that are far away from camera. Unity provides mechanisms for changing the level of detail (LOD) of objects according to their distance from camera. Up to three different models can be used for each object, with consecutively growing detail level. However, all three models must be imported to the project and added to a LOD group by the developer. [73]
- Testing different surface shader options for objects. There seemed to be clear drop in frame rate, when changing from the standard opaque shader to a more complicated one. However, no significant gain was observed when visualizing objects without textures or lightning.
- Adding occlusion culling to the scene. Occlusion culling means disabling (culling) of meshes, which are being hidden (occluded) by other objects. By default, all objects within camera's view angle are being rendered, even when the view is blocked by other objects in front of the camera. [74] In terminal environment this means, that all containers of a block are being rendered, despite that only a fraction of them is actually shown by the camera. The typical workflow for enabling occlusion culling involves selecting the meshes and generating (baking) the occlusion data in the editor during design-time [74]. Thus, applying it to a procedurally generated scene, where objects are being spawned during runtime, may be difficult.

- Using methods like global fog to limit the visible terminal area to a certain distance. This is more sophisticated method than bare camera clipping, which makes the distant objects disappear completely. However, camera clipping may be needed for culling the objects that are hidden behind the fog.

Even when using the listed techniques, keeping the frame rate constantly above 30 FPS may prove challenging, given that the whole terminal should be visible at all times. This is due to large amount of container objects, which cannot be simplified by much.

The last suggestion of hiding some of the objects is basically violating the principle of visualizing the whole terminal. One option could be to create UI mechanisms for toggling the fog on and off and changing the draw distances during runtime. This would let the user to decide the trade-off between the visibility and performance.

8.1.2 CPU profiling of back-end

It was not clear, whether the profiler data of Figure 32 included the data acquisition that takes place in the back-end. Back-end components were profiled separately by using the previously described test application and diagnostic tools provided by Visual Studio. The profiling was performed on a development laptop, powered by Intel Core i7-4810MQ with 2.80 GHZ clock rate. The resulting CPU load curve is presented in Figure 33.

The back-end seems to be consuming about 20% of all CPU resources, when running in Visual Studio debugger with logging output disabled. This number was, however, thought to be an overestimation caused by the debugging environment: When the project was running in Unity editor and camera was pointing away from containers and CHE, the whole process was consuming less than 4% of CPU resources according to Windows Task Manager. Since the back-end is not a separate process, its calculation load is supposed be included in that number.

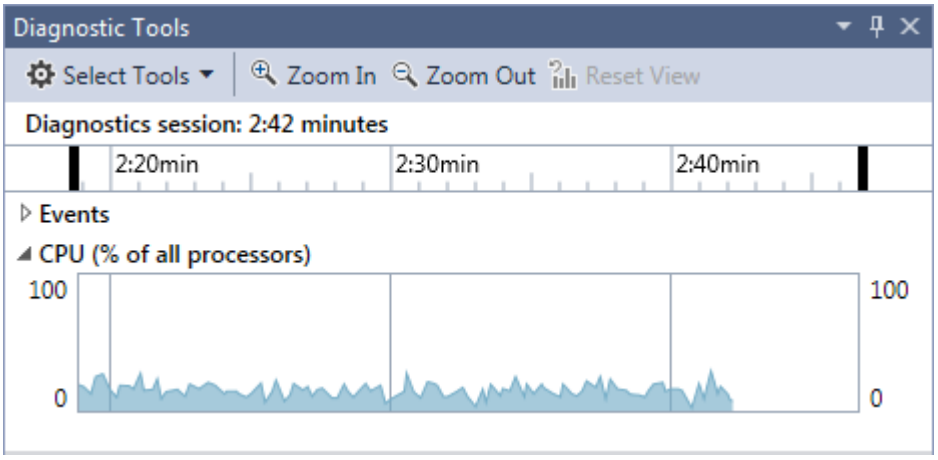

Figure 33: *CPU profiling of the back-end during the stress test*

8.1.3 Conclusion

Results of the performance test suggest that large amount of movement is not causing performance issues, when using the application with the specified amount of CHE and containers. The number of machines in the test was slightly smaller than was specified in R 22. However, the amount of movement tags per second was much larger than could be expected in real situations: In actual terminals, all the machines are rarely moving simultaneously, and all position tags of a machine are not changing at the same time. Generally, the performance test can be considered as a success, despite some shortcomings in the graphical performance.

8.2 Further development ideas

The implemented application satisfies all the most important functional and non-functional requirements from Chapter 5. Adding visualizations for further items – like routes, space reservations and new CHE types – should be straightforward, when expanding the existing framework of Unity scripts, cached objects and tag listeners.

The movement of CHE is currently animated on relatively coarse level: E.g. the wheels of machines are not moving, the lifting cables are not visualized, and there are no animations for twistlocks. There are currently no plans of adding any physics model to the application, as it would greatly increase the computational load. Thus, any movement that is not directly given by tag values must be approximated based on another events.

Some of the static objects from *terminal_layout.xml* are visualized as plain rectangular objects. More detailed models and additional program logic could be added to improve them as well.

8.2.1 Including interaction in the application

The application does not yet contain any interaction that was specified in R 8. A concept for adding message sending to the application is presented in Figure 34.

Classes of the diagram don’t correspond to any existing data structures in the application. It is e.g. not yet defined, whether there should be a separate cache for the items to be sent, or should they be attached to the relevant machines or control systems.

The message or command is supposed to be stored in the back-end until it is handled by the message handler thread of the communication library. The sending Unity script should not wait for any longer that is necessary to store the message in the back-end. This conforms to the principle of decoupling the communication components from the visualization, and vice-versa.

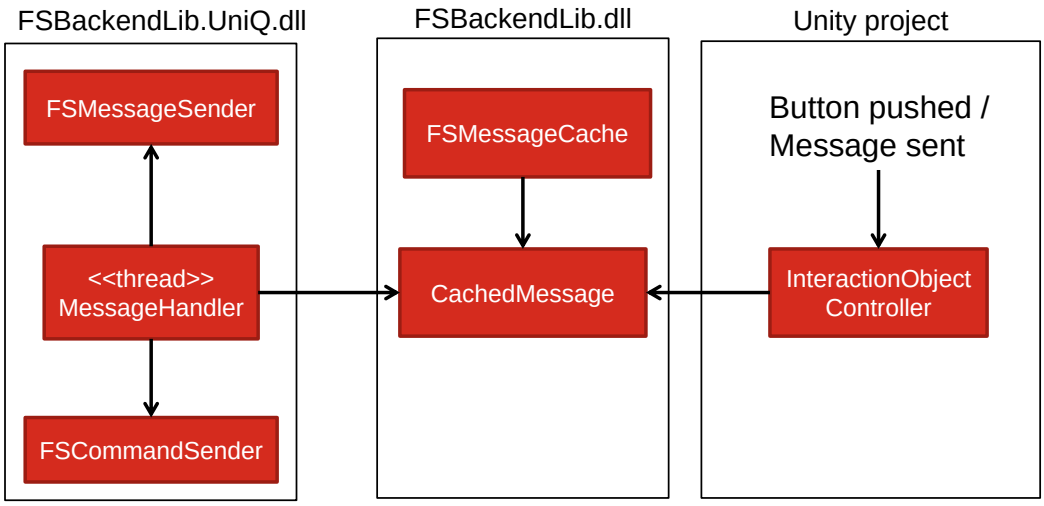

Figure 34: *Principle for adding message sending to the application*

The presented concept was tested briefly by adding few new methods to the Unity project and the back-end. The user was able to order deletion of a certain container by pressing F8 key in the GUI. Pressing of the key caused a variable in *FSContainerCache* to change its value. The variable was monitored by a thread owned by *FSCommunicationManager*, which converted the information into a message which was sent to UniQ. As a result, deletion of the container was observed in Fleetview and the 3D GUI alike. There was no observable delay or freezing associated with the event, proving that the message sending was decoupled from the visualization thread.

8.2.2 Alternative data sources

The application was originally specified to acquire its process data in real-time from a simulated TLS. There have also been ideas about adding playback functionality, which would allow replaying saved simulation results in various speeds. This could probably mean that the application should read the historical simulation data from a log file or database instead of receiving it from UniQ.

A principle for adding the log parsing into the existing application is presented in Figure 35. In this scheme, *FSBackendLib.UniQ* is replaced with a *LogPlayer* library, which is responsible for reading the logged simulation data and creating the corresponding update events to the cached objects at correct times. For the cached objects and the Unity project the situation would be the same as if the data was received from UniQ. This is one benefit from handling the data acquisition in a separate module.

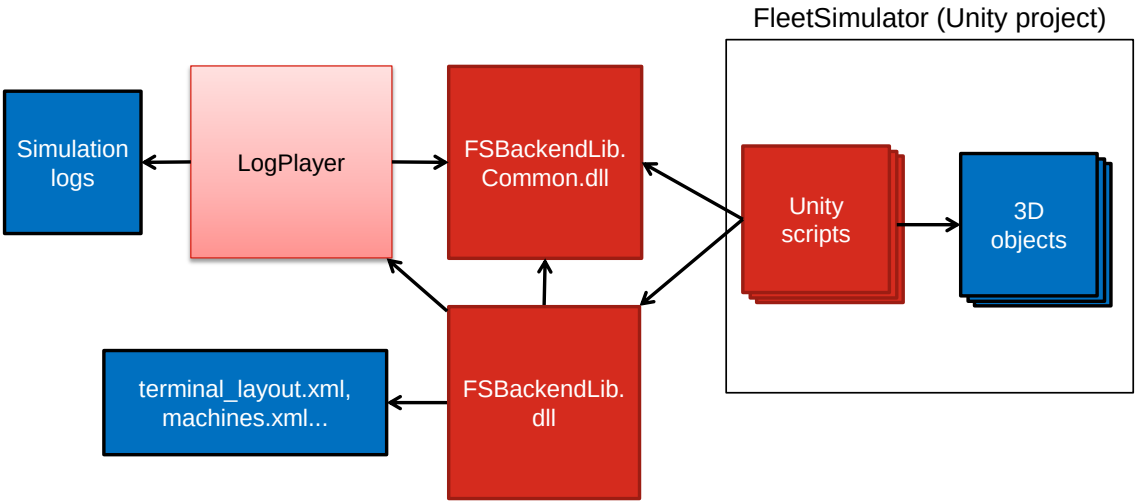

Figure 35: *Example of adding alternative data sources to the existing architecture*

For adding the playback functionality, some additional parametrization would be required, so that *FSSessionManager* may define the used data source. The Unity project should also have the necessary UI elements for controlling the playback, and there should be a mechanism for passing the commands to the *LogPlayer* library.

8.2.3 Other requirements

Most items in the low priority part of the requirements list are related to features offered in the 3D GUI. They don't add any complexity to the back-end, as they could be easily implemented by using the existing architecture and the interaction system described above.

R 14 states that it should be possible to send commands from the 3D GUI to Fleetview or some other application to get status information for machines. It has not yet been studied, if such cross-application linking can be implemented. It is, however, perfectly possible to run the application side-by-side with Fleetview to see the CHE status data together with 2D and 3D visualizations.

R 16 states that it should be possible to change the used 3D models or add new ones in the built application. This would require some import mechanism to be added to the application itself, as Unity does not store the models in the application folder as separate files. A page in Unify Wiki provides source code for *ObjImporter* class, which could be used to create meshes from .obj-files during runtime [75]. In addition to the meshes, at least textures would be needed for creating a visible model. Runtime model generation should certainly be possible, but it would be relatively complicated in comparison to handling the models it in the editor prior to the build.

9. CONCLUSIONS

In this thesis, Unity game engine was used to develop a real-time 3D visualization tool for automated container terminals. The implemented application can be configured by using terminal-specific XML files and it visualizes the static terminal layout and container handling machines based on the XML data. Container handling operations and machine movements are animated based on live data from the process. The application communicates with the terminal automation system by using the same communication protocol that is used in actual production systems. The application can be used to visualize simulated systems and actual terminals alike. It was also proven, that the application can be extended to send messages back to the terminal automation system, acting as an interactive GUI.

The visualization was tested with a virtual terminal, containing several thousand containers and 47 individual machines. It was proven that the application is able to acquire, store and visualize machine position information at rate of hundreds of messages per second without showing any significant flaws in performance. However, the large amount of objects in the terminal makes the visualization of the whole area a challenging task for a conventional PC. Further graphical optimization and experimenting is required in order to provide sufficient frame rate and smooth animation in all situations.

According to the performed literature survey, 3D visualizations and game engine based experiments play an increasingly important role in many fields. The wide availability of game engines has made 3D application creation more accessible than ever, with Unity being by far the most used platform at the moment. In the field of container terminal logistics, many commercial 3D applications are already provided for planning, simulation and supervision of operations. The emerging virtual reality technologies are also bringing new and exciting possibilities e.g. for creating immersive training simulations for scenarios, that would otherwise be very difficult to emulate.

Unity was found to be an excellent tool for creating 3D applications with relatively little learning effort. It can be recommended for visualization projects, despite some technical shortcomings, like outdated .NET version. Unity may not be the most graphically advanced game engine in the market, nor is it necessary the best platform for performing heavy calculations: In this thesis project, the focus was on visualizing a large-scale process, which is being simulated in a separate environment. When creating accurate real-time simulations for objects with complicated physics, a totally different set of requirements would apply for the development tools. Running such simulations for a large number of objects might also be impossible on a single computer.

The developed visualization tool has many possible applications within Kalmar, as it can be connected to any system that implements the common communication protocol. Possible fields of application include product development, marketing, sales, project engineering, system testing and possibly even adding the visualization tool in the customer software packages. Due to the modular structure of the software, new features and alternative data sources can be added without changing the internal data model or visualization logic. Development of the application will continue within Kalmar and it will be used for producing visualizations for upcoming terminal emulation projects.

REFERENCES

[1] J. Bose, T. Reiners, D. Steenken, S. Voss, Vehicle dispatching at seaport container terminals using evolutionary algorithms, System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on, pp. 10 pp. vol.2.

[2] D. Steenken, S. Voß, R. Stahlbock, Container terminal operation and operations research - a classification and literature review, in: H. Günther, K.H. Kim (ed.), Container Terminals and Automated Transport Systems, Springer Berlin Heidelberg, 2005, pp. 3-49.

[3] Container port throughput, annual, 2008-2014, UNCTADstat, web page. Available (accessed 20.07.2016): http://unctadstat.unctad.org/wds/TableViewer/tableView.aspx?ReportId=13321.

[4] Container port traffic, The World Bank Group, web page. Available (accessed 20.07.2016): http://data.worldbank.org/indicator/IS.SHP.GOOD.TU.

[5] C.A. Thoresen, Container Terminals, in: Port Designer's Handbook (3rd Edition), ICE Publishing, 2014, pp. 321-353.

[6] B. Brinkmann, Operations Systems of Container Terminals: A Compendious Overview, in: J. Böse (ed.), Handbook of Terminal Planning, Springer-Verlag New York, 2011, pp. 25-39.

[7] Equipment, Kalmar, web page. Available (accessed 14.07.2016): https://www.kalmarglobal.com/equipment/.

[8] J. Pirhonen, Automated Shuttle Carrier Concept, in: J. Böse (ed.), Handbook of Terminal Planning, Springer-Verlag New York, 2011, pp. 41-59.

[9] Internal media bank, Kalmar, Copyright © Cargotec 2014.

[10] M. Soukka, Area Layout Data Handling Solution for Container Terminals, Master of Science thesis, Tampere University of Technology, 2015, 62 p. Available: http://urn.fi/URN:NBN:fi:tty-201512181835.

[11] Kalmar Shuttle Carrier - Data sheet, Kalmar, web page. Available (accessed 19.07.2016): https://www.kalmarglobal.com/globalassets/equipment/shuttle-carriers/kalmar-shuttle-carrier-data-sheet-2016.pdf.

[12] Innovation makes TraPac's moves count, Kalmar, web page. Available (accessed 19.07.2016): https://www.kalmarglobal.com/globalassets/customer-cases/all-customer-cases/trapac/customer-case-trapac.pdf.

[13] F. Meisel, Operational Planning Problems, in: Seaside Operations Planning in Container Terminals, Physica-Verlag, Heidelberg, Germany, 2009, pp. 17-30.

[14] N4 - Features & Functionality, Navis, web page. Available (accessed 28.07.2016): http://navis.com/sites/default/files/pages/docs/n4_features_and_functionality_3.1_0_2015.pdf.

[15] Manage any volume of containers with Master Terminal, Jade Software Corporation Limited, web page. Available (accessed 28.07.2016): https://www.jadeworld.com/solutions-for/logistics/master-terminal-tos/containers/.

[16] RBS website, Realtime Business Solutions, web page. Available (accessed 28.07.2016): http://www.rbs-emea.com/.

[17] Kalmar Automation System Architecture, Cargotec Finland Oy, Internal documentation, 2014.

[18] Terminal UI, Cargotec Finland Oy, Training material, 2016.

[19] T. Piipari, Dynamic Configuration Management, Master of Science thesis, Tampere University of Technology, 2013, 44 p. Available: http://urn.fi/URN:NBN:fi:tty-201303211098.

[20] A. Lehtonen, Data Distribution Protocol for Container Handling Equipment Interface, Master of Science thesis, Tampere University of Technology, 2010, 82 p.

[21] The Spread Toolkit, Spread Concepts LLC, web page. Available (accessed 03.08.2016): http://www.spread.org/.

[22] Hannu Santahuhta, Senior Manager, Simulation & Virtual Environments, Kalmar, Interview 05.08.2016.

[23] Internal marketing material, Kalmar, Copyright © Cargotec 2016.

[24] DreamPort User Guide, Cargotec Corporation, 2016.

[25] FlexTerm webpage, Moffat & Nichol, web page. Available (accessed 15.08.2016): http://www.flexterm.com/index.html.

[26] FlexTerm / FlexSim Container Terminal Software, Talumis BV, web page. Available (accessed 15.08.2016): http://talumis.com/flexterm/.

[27] Terminal View product sheet, Tideworks Technology Inc, web page. Available (accessed 15.08.2016): https://www.tideworks.com/wp-content/uploads/2015/03/Terminal-View-product-sheet-120914.pdf.

[28] Tideworks charts its path, WorldCargo News, Vol. November, 2013, pp. 30-31.

[29] CHESSCON Overview, ISL Applications GMBH, web page. Available (accessed 15.08.2016): http://www.downloads.isl-applications.com/CHESSCON-Overview.pdf.

[30] CHESSCON Virtual Terminal, ISL Applications GMBH, web page. Available (accessed 15.08.2016): http://www.downloads.isl-applications.com/CHESSCON-VirtualTerminal.pdf.

[31] TBA CONTROLS web page and brochure, TBA Netherlands, web page. Available (accessed 15.08.2016): https://www.tba.nl/en/software/controls.

[32] J. L. Bijl, C. A. Boer, Advanced 3D visualization for simulation using game technology, Proceedings of the 2011 Winter Simulation Conference (WSC), 11-14 December 2011, IEEE, pp. 2810-2821.

[33] Y. Saanen, M. Koekoek, The Future: Serious Gaming in Automated Terminals, Port Technology, No. 65, February 2015, pp. 81-83.

[34] Play the terminal: 'Safe-T Game', TBA Netherlands, web page. Available (accessed 15.08.2016): https://www.tba.nl/en/company-info/press-section/105/play-the-terminal-safe-t-game/.

[35] H. Lau, L. Chan, R. Wong, A virtual container terminal simulator for the design of terminal operation, International Journal on Interactive Design and Manufacturing (IJIDeM), Vol. 1, No. 2, 2007, pp. 107-113.

[36] A. Bruzzone, F. Longo, L. Nicoletti, R. Diaz, Virtual simulation for training in ports environments, Proceedings of the 2011 Summer Computer Simulation Conference (SCSC '11), Society of Modeling & Simulation International, Vista, CA, USA., pp. 235-242.

[37] D. Yuan, X. Jin, J. Zhang, D. Han, Applying Open Source Game Engine for Building Visual Simulation Training System of Fire Fighting, AsiaSim 2007: Asia Simulation Conference 2007, Seoul, Korea, October 10-12, 2007. Proceedings, Springer-Verlag Berlin, Heidelberg, Germany, pp. 365-374.

[38] L. Chittaro, Serious Games for Training Occupants of a Building in Personal Fire Safety Skills, 2009 Conference in Games and Virtual Worlds for Serious Applications, IEEE, pp. 76-83.

[39] A. Mól, C. Jorge, P. Couto, Using a Game Engine for VR Simulations in Evacuation Planning, IEEE Computer Graphics and Applications, Vol. 28, No. 3, 2008, pp. 6-12.

[40] A. Peltola, Virtuaalitodellisuuden soveltaminen opetus- ja opastuskäytössä, Bachelor's thesis, Turku University of Applied Sciences, 2015, 42 p. Available: http://urn.fi/URN:NBN:fi:amk-2015111916864.

[41] J. Kaasalainen, Virtuaalitodellisuuslaitteiden sovellukset kuntoutuksessa ja havainnoinnin analysoinnissa: suunnittelu ja toteutus, Bachelor's thesis, Turku University of Applied Sciences, 2015, 40 p. Available: http://urn.fi/URN:NBN:fi:amk-201505219335.

[42] T. Tapio, Ajosimulaation toteutus Unity 3D-pelimoottorilla, Bachelor's thesis, Lapland University of Applied Sciences, 2014, 65 p. Available: http://urn.fi/URN:NBN:fi:amk-201404234724.

[43] S. Marks, J. Windsor, B. Wünsche, Evaluation of Game Engines for Simulated Clinical Training, Proceedings of the New Zealand Computer Science Research Student Conference 2008, The University of Auckland, Auckland, New Zealand, pp. 92-99.

[44] M. Kibsgaard, K.K. Thomsen, M. Kraus, Simulation of surgical cutting in deformable bodies using a game engine, Abstract, GRAPP 2014 - Proceedings of the 9th International Conference on Computer Graphics Theory and Applications, 7-9 January 2014, pp. 342-347.

[45] J. Korkiakoski, Virtuaalilasien käyttö Unity-pelimoottorilla rakennetussa 3D-visualisointisovelluksessa, Bachelor's thesis, Oulu University of Applied Sciences, 2016, 32 p. Available: http://urn.fi/URN:NBN:fi:amk-201604124219.

[46] J. Wang, L. Phillips, J. Moreland, B. Wu, C. Zhou, Simulation and Visualization of Industrial Processes in Unity, SummerSim '15 Proceedings of the Conference on Summer Computer Simulation, July 26-29, 2015, Society for Modeling & Simulation International (SCS), San Diego, CA, USA, pp. 1-7.

[47] H. García Pájaro, A 3D Real-Time Monitoring System for a Production Line, Master of Science Thesis, Tampere University of Technology, 2012, 78 p. Available: http://urn.fi/URN:NBN:fi:tty-201206181197.

[48] T. Makkonen, R. Heikkilä, A. Kaaranka, K. Nevala, Roadwork site 3D virtual visualization using open source game engine and open information transfer, Abstract, 31st International Symposium on Automation and Robotics in Construction and Mining, ISARC 2014 - Proceedings, 9-11 July 2014, University of Technology Sydney, pp. 697-701.

[49] M. Korpioksa, Cooperation between Unity and PLC, Bachelor's thesis, Seinäjoki University of Applied Sciences, 2014, 42 p. Available: http://urn.fi/URN:NBN:fi:amk-2014120318107.

[50] T. Patana, Raepuhallusrobotin simulointiympäristö, Bachelor's thesis, Kajaani University of Applied Sciences, 2012, 31 p. Available: http://urn.fi/URN:NBN:fi:amk-2012060712105.

[51] B. Nilson, M. Söderberg, Game Engine Architecture, Mälardalen University, 2007, Available: http://www.idt.mdh.se/kurser/cd5130/jgms/2007lp4/report9.pdf.

[52] M. Enger, Game Engines: How do they work? CBS Interactive Inc., web page. Available (accessed 08.10.2016): http://www.giantbomb.com/profile/michaelenger/blog/game-engines-how-do-they-work/101529/.

[53] E. Kalderon, Game engines: What they are and how they work, web page. Available (accessed 08.10.2016): https://nullpwd.wordpress.com/2011/05/09/game-engines-what-they-are-and-how-they-work/.

[54] J. Brodkin, How Unity3D Became a Game-Development Beast, Dice / DHI Group, Inc., web page. Available (accessed 20.08.2016): http://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/.

[55] B. Steiner, How the Unreal Engine Became a Real Gaming Powerhouse, Hearst Communications, Inc., web page. Available (accessed 20.08.2016): http://www.popularmechanics.com/culture/gaming/a9178/how-the-unreal-engine-became-a-real-gaming-powerhouse-15625586/.

[56] C. Bleszinski, History of the Unreal Engine, IGN / Ziff Davis LLC., web page. Available (accessed 20.08.2016): http://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine.

[57] This engine is dominating the gaming industry right now, The Next Web, Inc., web page. Available (accessed 20.08.2016): http://thenextweb.com/insider/2016/03/24/engine-dominating-gaming-industry-right-now/.

[58] Unity - Multiplatform - Publish your game to over 10 platforms, Unity Technologies, web page. Available (accessed 20.08.2016): https://unity3d.com/unity/multiplatform.

[59] Unreal Engine FAQ, Epic Games, Inc., web page. Available (accessed 09.10.2016): https://www.unrealengine.com/faq.

[60] Licensing options of Unity, Unity Technologies, web page. Available (accessed 09.10.2016): https://store.unity.com/?_ga=1.155460989.770757390.1454678151.

[61] Unreal Engine End User License Agreement, Epic Games, Inc, web page. Available (accessed 09.10.2016): https://www.unrealengine.com/eula.

[62] Unreal Engine 4 For Unity Developers, Epic Games, Inc., web page. Available (accessed 11.10.2016): https://docs.unrealengine.com/latest/INT/GettingStarted/FromUnity/.

[63] Unity manual: 3D formats, Unity Technologies, web page. Available (accessed 11.10.2016): https://docs.unity3d.com/Manual/3D-formats.html.

[64] Unity manual: Textures, Unity Technologies, web page. Available (accessed 11.10.2016): https://docs.unity3d.com/Manual/class-TextureImporter.html.

[65] Unity manual: Creating and Using Scripts, Unity Technologies, web page. Available (accessed 12.10.2016): https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html.

[66] Unity scripting reference: MonoBehaviour, Unity Technologies, web page. Available (accessed 12.10.2016):
https://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

[67] Blueprints Visual Scripting - Functions, Epic Games, Inc., web page. Available (accessed 13.10.2016):
https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Functions/.

[68] What is the cost of using Blueprint instead of C++?, UE4 AnswerHub, web page. Available (accessed 13.10.2016):
https://answers.unrealengine.com/questions/23167/blueprint-overhead.html.

[69] A. Koloska, Blueprints vs C++, web page. Available (accessed 13.10.2016):
http://shootertutorial.com/2015/06/06/blueprints-vs-c/.

[70] A. Biggs, Unity Devs, stop using GameObject.Find! web page. Available (accessed 30.09.2016): https://akbiggs.silvrback.com/please-stop-using-gameobject-find.

[71] Unity scripting reference: GameObject.Find, Unity Technologies, web page. Available (accessed 30.09.2016):
https://docs.unity3d.com/ScriptReference/GameObject.Find.html.

[72] Unity manual: Optimizing graphics performance, Unity Technologies, web page. Available (accessed 02.10.2016):
https://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html.

[73] Unity manual: LOD Group, Unity Technologies, web page. Available (accessed 02.10.2016): https://docs.unity3d.com/Manual/class-LODGroup.html.

[74] Unity manual: Occlusion Culling, Unity Technologies, web page. Available (accessed 06.10.2016): https://docs.unity3d.com/Manual/OcclusionCulling.html.

[75] ObjImporter, Unify Wiki, web page. Available (accessed 02.10.2016):
http://wiki.unity3d.com/index.php?title=ObjImporter.

APPENDIX 1: SCREENSHOTS FROM THE APPLICATION

When the application is started, it will first open a configuration window, which has separate tabs for graphics and user input. These are illustrated in Figure 36 and Figure 37 respectively.

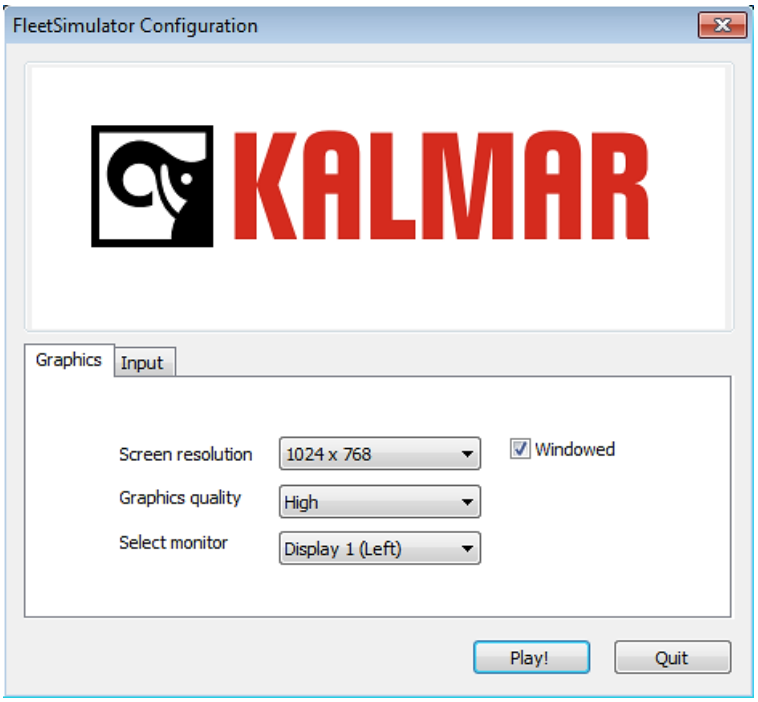

Figure 36: *Initial configuration for graphics*

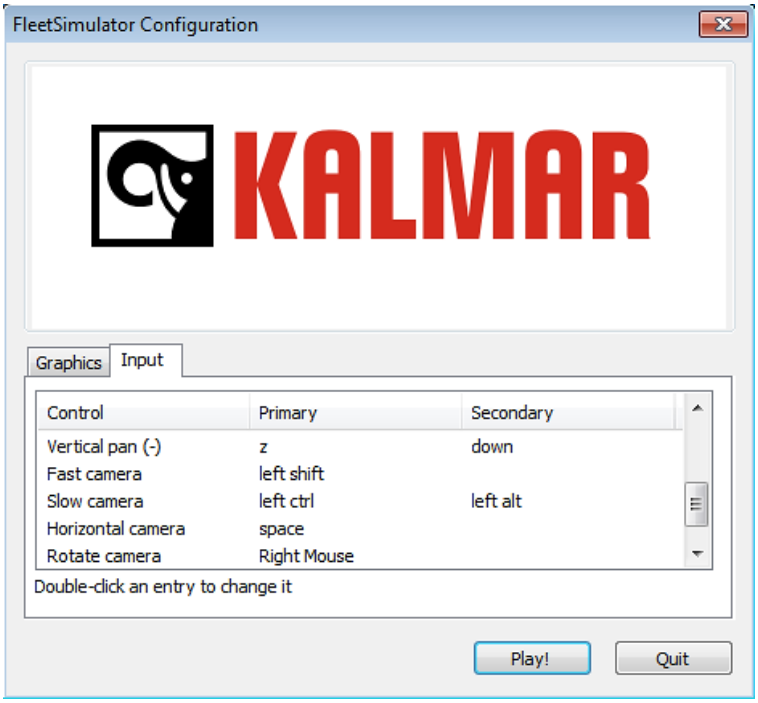

Figure 37: *Initial configuration for input devices*

The configuration window is created automatically by Unity during the build, if the developer has not chosen to disable it. Its contents are set according to the settings defined in the Unity project. It is not possible to add any custom content besides the ones that are included by default. Thus, for any additional configuration, a separate configuration view or other mechanism needs to be added to the application itself. This was one of the main reasons for using separate XML file for configuration.

After proceeding from the configuration window, the application starts in an initialization view, which is illustrated in Figure 38. Parsing of the configuration files and loading of the chosen target scene takes place at this point. The scene is selected according to the *StaticLayoutType* parameter in *FleetSimulator.config*.

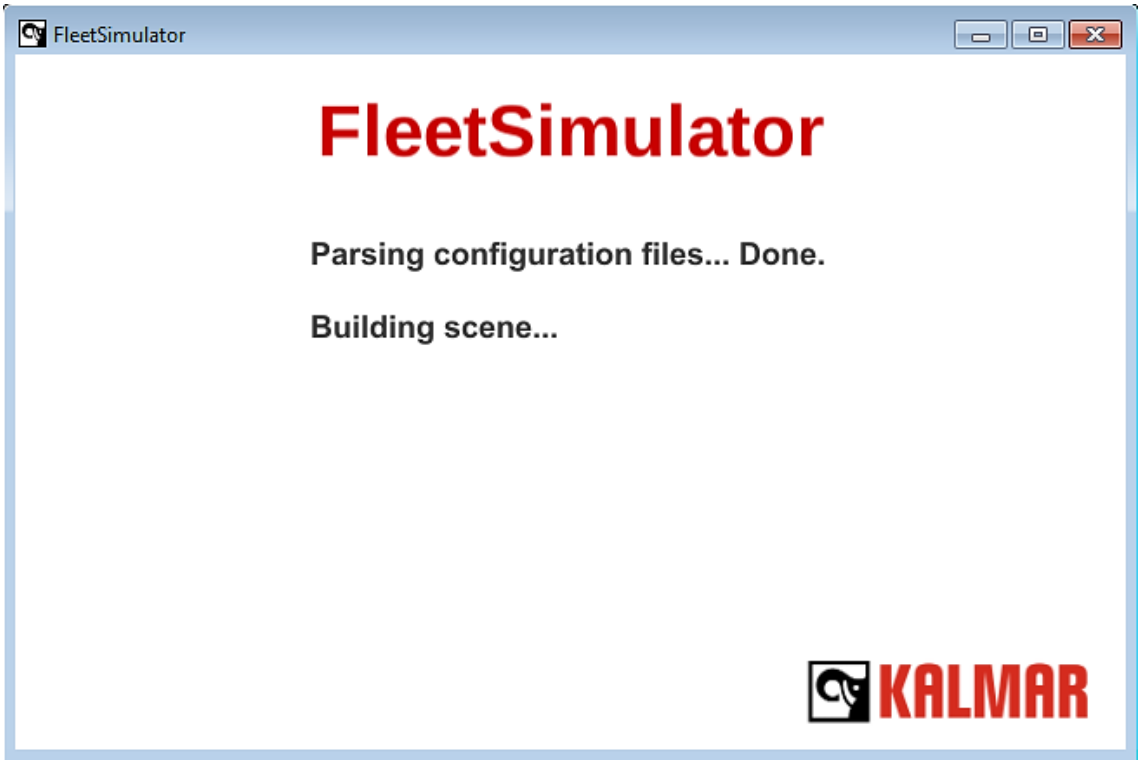

Figure 38: *Initialization view*

When loading of the target scene is completed, the application opens a 3D view of the terminal, where the user can navigate freely with a flying camera. The machines are first visualized in their default positions and moved immediately, when their corresponding position tags are received.

Container objects are created one-by-one during runtime. Container spawn rate is limited to 200 boxes per second in order to avoid excessive CPU load and keep the UI navigable during the initialization. Full initialization of a terminal with 10,000 containers takes approximately one minute.

Figure 39 shows an aerial overview of the simulated example terminal as visualized in the application. Figure 40 shows the same area visualized in Fleetview.

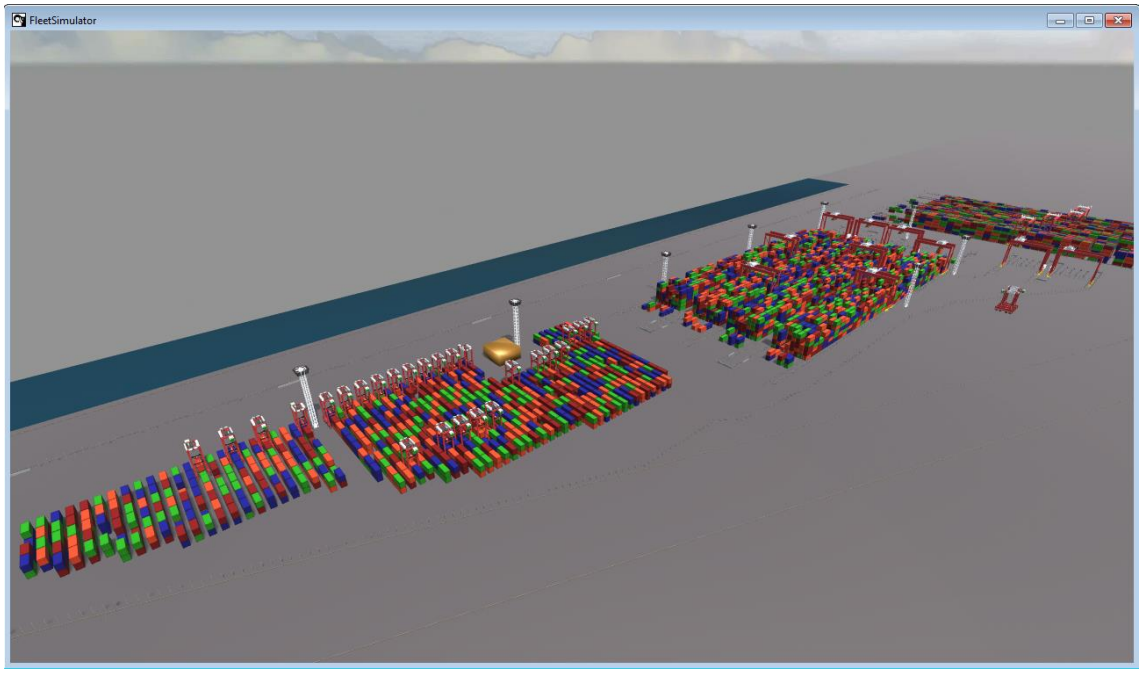

Figure 39: *Aerial 3D view of the simulated example terminal*

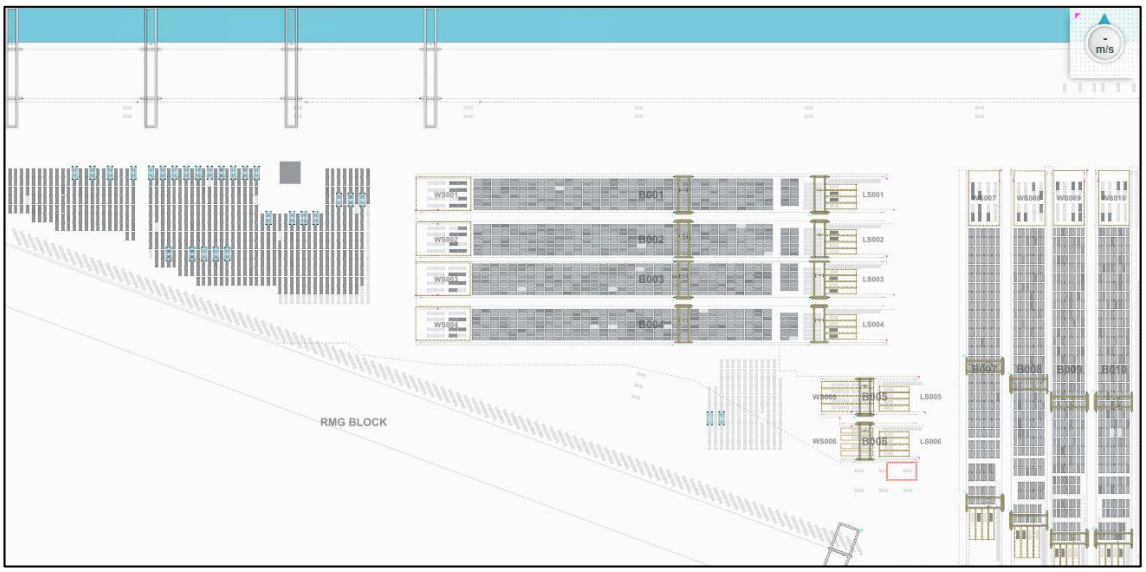

Figure 40: *The simulated example terminal visualized in Fleetview*

Figure 41 and Figure 42 present closer views of ASC cranes. Straddle and shuttle operations are illustrated in Figure 43 and Figure 44 respectively.

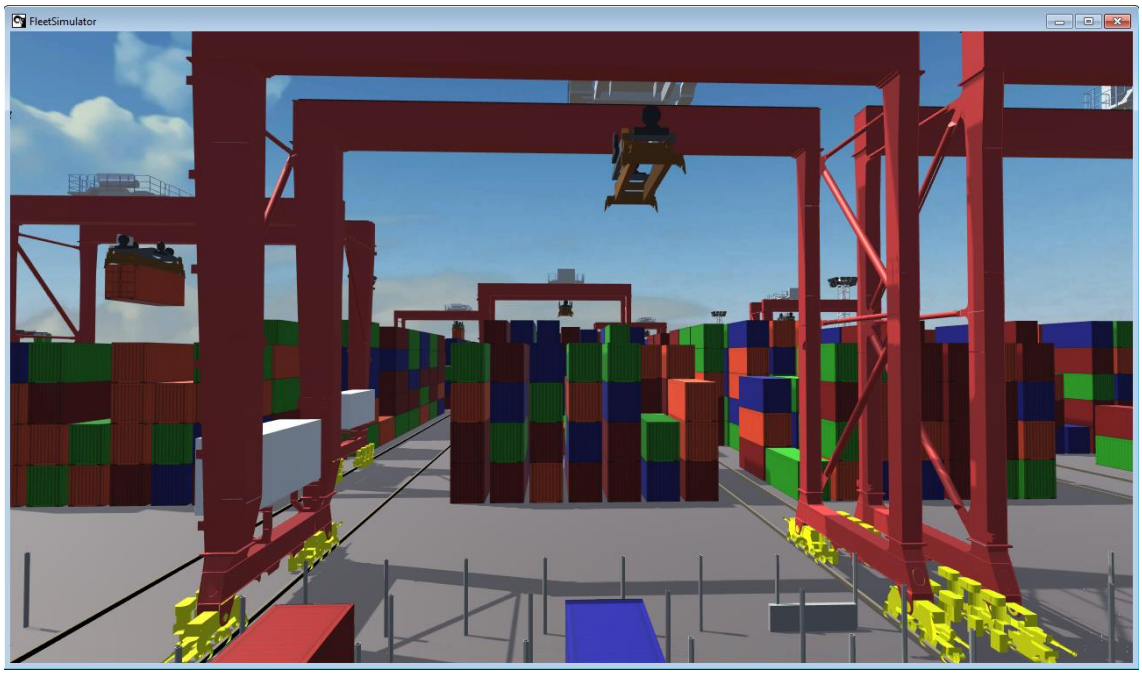

Figure 41: *First-person view of ASC blocks*

Figure 42: *ASC crane picking a container*

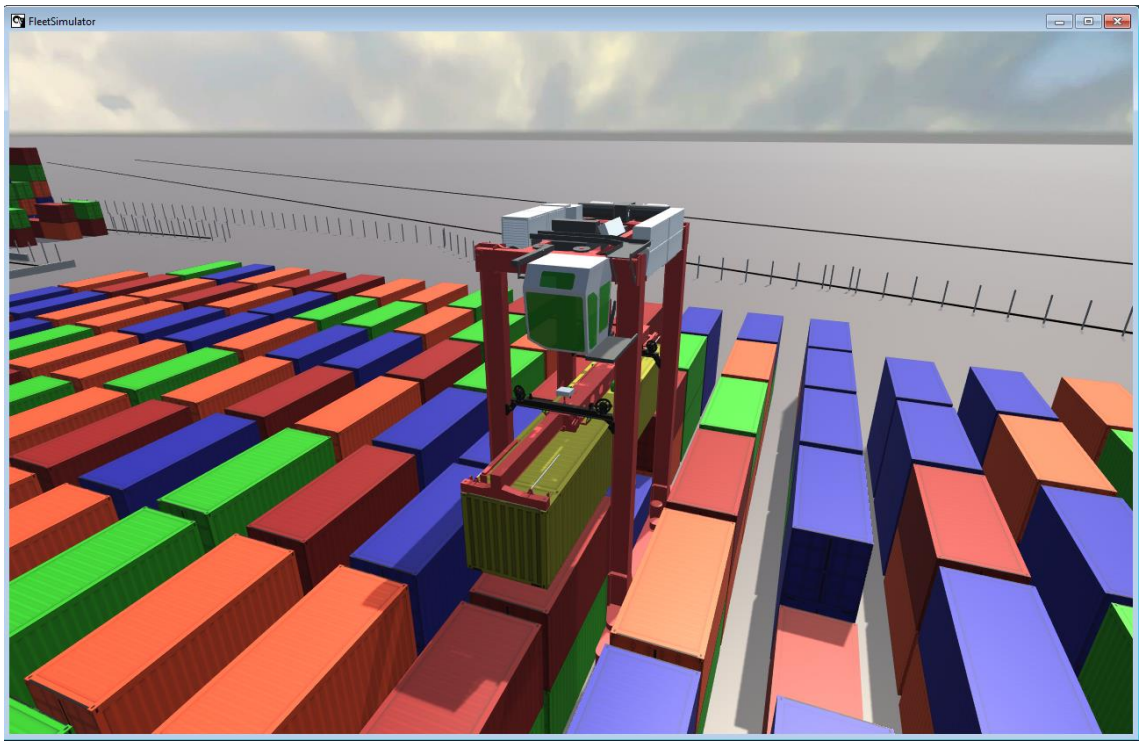

Figure 43: *Straddle carrier navigating between linear stacks*

Figure 44: *Shuttle carrier bringing a container to the interchange area*

The previous pictures illustrated the *dynamic* terminal layout, which is created in runtime according to *terminal_layout.xml*. In addition to the dynamic layout, and the Rusko test site illustrated in Figure 24, the application offers two more options for the static terminal layout. The *empty* layout (Figure 45) contains a pre-made scene with

fixed field dimensions and background. The *none* layout (Figure 46) contains no static objects at all, which makes the animations look somewhat smoother than in other scenes due to higher frame rate.

Figure 45: *Machines and containers visualized in the "empty" layout scene*

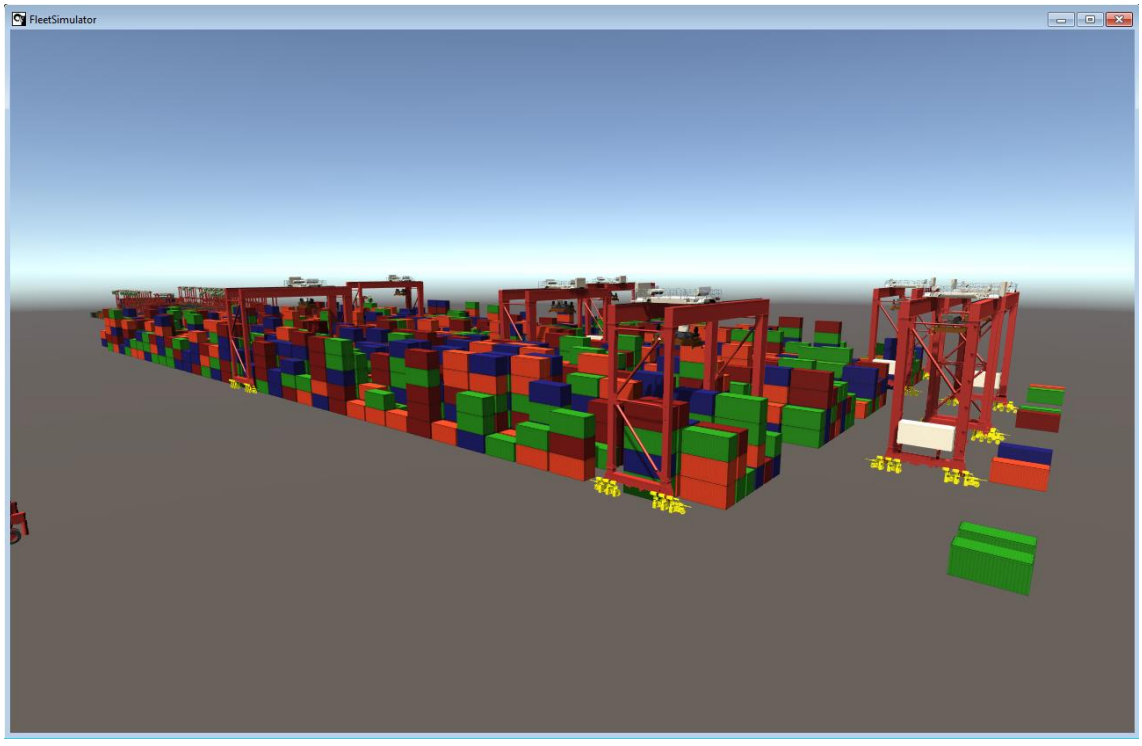

Figure 46: *Machines and containers visualized in the "none" layout scene*