TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Hendrik Laas 176576IAPM

WEBASSEMBLY JA JAVASCRIPTI JÕUDLUSE ANALÜÜS GRAAFIKUTE JOONISTAMISE NÄITEL

magistritöö

Juhendaja: Martin Verrev
MSE

Tallinn 2019

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Hendrik Laas

07.05.2019

Annotatsioon

Käesoleva töö peamine eesmärk on uurida, kas ja kui palju võimaldab WebAssembly kaasamine graafikute loomist veebirakendustes kiirendada erineva mahuga andmestikkude korral. Töö lisaeesmärgiks on kirjeldada töö käigus kujunenud töövoog ning selgitada, milline on optimaalne töövoog WebAssembly rakenduse arendamisel. Töö eesmärkide täitmiseks luuakse töö käigus graafikute loomise teek veebirakendustele, milles arvutusi nõudvad osad on loodud WebAssembly's ning HTML elementide manipuleerimiseks vajalikud osad JavaScriptis.

Eesmärkide saavutamiseks dubleeriti WebAssembly's kirjutatud osad ka JavaScriptis ning viidi läbi võrdlusanalüüs kahe lahenduse vahel. Realiseeriti 2 graafikutüüpi: joondiagramm ning tellimusraamatu sügavuse graafik. Graafikute võrdlusanalüüsil võrreldi graafikute joonistamise kiirust ning andmete edastamise kiirust. Võrdlusanalüüs viidi läbi erineva suurusega andmestikkude korral ning kolme erineva seadme peal, millest kõigis oli kasutusel 3 erinevat veebibrauserit.

Võrdlusanalüüsist selgus, et WebAssembly'ga lahenduse puhul toimib graafiku joonistamine kiiremini, kuid andmete edastamine aeglasemalt. Näiteks joondiagrammi joonistamisel 500000 andepunkti korral oli WebAssembly'ga lahendus 2.91 korda kiirem ning tellimusraamatu sügavuse graafiku korral 50000 – 52000 andmepukti korral 5.02 korda kiirem. Andmete edastamine 500000 korral joondiagrammile oli JavaScriptiga lahendus 3.33 korda kiirem. Sisuliselt võib väita, et graafiku esmane kuvamine kasutajale on JavaScriptis realiseeritud lahenduse puhul kiirem, kuid iga järgneb joonistamine on kiirem WebAssembly kasutamise puhul.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 52 leheküljel, 11 peatükki, 18 joonist, 11 tabelit.

Abstract

Performance Analysis of WebAssembly and JavaScript Based on Drawing Charts

The main goal of the present thesis is to analyse if and how much does using WebAssembly accelerate drawing charts in web applications for data sets of different sizes. The side goal of this thesis is to describe development flow that formed during work, describe its faults and to propose what could be done better. Charting library for web applications is developed during the thesis. All parts that need heavy calculations are developed in WebAssembly and the parts that need HTML elements manipulations are developed in JavaScript.

To fill the goal, parts developed in WebAssembly are duplicated in JavaScript and benchmark between both solutions is carried out. Two types of charts were developed: line chart and orderbooks depth chart. In benchmark, the speed of drawing the charts was measured, as well as sending data to chart. Benchmark was carried out using datasets of different sizes and on 3 different devices, all of which had 3 web browsers.

Results from benchmarking stated that using WebAssembly for calculations, the speed of drawing the chart was smaller, but on the other hand the speed of sending data to chart was greater. For example drawing the line chart for 500000 data points the solution with WebAssembly performed 2.91 times faster and drawing the orderbook depth chart for 50000 – 52000 data points performed 5.02 times faster for solution with WebAssembly. Sending 500000 data points to line chart performed 3.33 times faster for solution with JavaScript. It could be stated that the first display of chart is achieved faster for solution with JavaScript but every other draw is faster for solution with WebAssembly.

The thesis is in Estonian and contains 52 pages of text, 11 chapters, 18 figures, 11 tables.

Lühendite ja mõistete sõnastik

API	*Application programming interface*, rakendusliides
ARRAYBUFFER	*In JavaScript ArrayBuffer object is used to represent a generic, fixed-length raw binary data buffer*, JavaScriptis kasutatav objekt, mis kujutab endas üldist fikseeritud pikkusega binaarset andmepuhvrit
BCH	Bitcoin Cash
BTC	Bitcoin
BTCUSD	Kauplemispaar Bitcoin – Ameerika dollar
CSS	*Cascading Style Sheets*, kaskaadlaadistik
CSV	*Comma-separated values*, komaga eraldatud väärtused
DOM	*Document Object Model*, dokumendi objektimudel
ETHUSD	Kauplemispaar Ethereum – Ameerika dollar
FETCH	*JavaScript API that API provides an interface for fetching resources (including across the network)*, JavaScripti API, mida saab kasutada päringute tegemiseks (kaasa arvatud üle võrgu)
HTML	*Hypertext Markup Language*, hüperteksti märgistuskeel
JSON	*JavaScript Object Notation*, JavaScripti objekti notatsioon
LITTLE-ENDIAN	*An order in which the "little end" (least significant value in the sequence) is stored first in computer memory*, järjestus, mille puhul kõige vähem oluline väärtus on arvuti mällu salvestatud esimesena
PROMISE	*Object that represents the eventual completion (or failure) of an asynchronous operation, and its resulting value*, objekt, mis esitab asünkroonse operatsiooni õnnestumist (või ebaõnnestumist) ja selle tulemust
SIMD	*Single instruction, multiple data*, tehnoloogia, mille puhul samu instruktsioone täidetakse paralleelselt mitme andmepunkti peal
SVG	*Scalable Vector Graphics*, skaleeruv vektorgraafika

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Seoses erinevate nutiseadmete arvu kasvuga on kasvanud vajadus rakenduste üleviimiseks veebibrauseritesse, et need oleks võimalikult lihtsasti kättesaadavad igas seadmes. Kuna JavaScripti operatsioonikiirus ei võimalda kõiki rakendusi veebibrauserisse üle viia, siis on välja töötatud uusi tehnoloogiaid. Üheks uueks tehnoloogiaks on WebAssembly. WebAssembly on binaarne käsuformaat, mis võimaldab kompileerida kõrgkeeli (C, C++, Rust) veebirakendustesse. WebAssembly on sisse ehtitatud veebibrauserite uuematesse versioonidesse ning see on kasutatav koos JavaScriptiga, mis tähendab, et WebAssembly ei ole mõeldud JavaScripti asendamiseks, vaid selle täiendamiseks.

Üks moodus WebAssembly kasutamiseks veebirakendustes on kõige rohkem arvutusi nõudvad osad realiseerida WebAssembly's ning ülejäänud JavaScriptis. Sellise arhitektuuriga rakendus luuakse ka selle töö käigus. Loodavaks rakenduseks on graafikute joonistamise teek veebirakendusetele, mille arvutusi nõudvad osad luuakse WebAssembly's ning HTML elementide juhtimine toimub läbi JavaScripti. Arhitektuuri teeb võimalikus JavaScripti ja WebAssembly vaheline lihtne ja efektiivne suhtlus. Töö käigus realiseeritakse ajaseeriatel põhinev mitut joondiagrammi toetav graafik ning tellimusraamatu sügavuse graafik.

Suure hulga andmete veebibrauseris esitamisel võtavad andmeteisendused ja nende kuvamine JavaScriptis piisavalt aega, et tekiksid viivitused graafikute renderdamisel. Viivitused tekivad enamasti suurte andmestikkude kuvamisel. Näiteks andes olemasolevale teegile AmCharts ette 20000 andmepunkti, kulus graafiku loomiseks 10 testi korral keskmiselt 6.83 sekundit. Nõnda suured andmehulgad on tavaliselt ajas muutuvad andmed - näiteks krüptorahade väärtuse muutus perioodi jooksul. Viivitused suurenevad kui graafikul kuvada korraga mitut andmestikku, näiteks mitu joondiagrammi ühel graafikul.

Töö eesmärgiks on uurida, kas ja kui palju võimaldab WebAssembly kaasamine graafikute loomist veebirakendustes kiirendada erineva suurusega andmestikkude korral.

Töö käigus realiseeritakse graafikute loomise teek veebirakendustele kasutades WebAssembly’t ning sama funktsionaalsus dubleeritakse ka JavaScriptis, et võrdlusanalüüs läbi viia. Töö käigus saadud tulemused dokumenteeritakse ja tulemuste põhjal tehakse järeldusi, samuti pakutakse välja olukorrad, mille puhul WebAssembly kasutusele võtmine kasuks tuleb ning, mille puhul mitte. Töö käigus kujunenud töövoog kirjeldatakse ning selgitatakse, mida saaks teha paremini, kuna WebAssembly on uus tehnoloogia ning optimaalset töövoogu arendamiseks ei ole üheselt välja kujunenud.

Töö motivatsiooniks on andmete puudumine selle kohta, kuidas WebAssembly ja JavaScripti koostöös veebirakenduste kiirused erinevad puhta JavaScriptiga lahendustest. Lisaks ei ole uuritud kiiruse erinevusi erineva suurusega andmestikkude korral. WebAssembly kiirust võrreldes JavaScriptiga on küll uuritud mõnes eelnevas uurimuses, kuid nendes uurimustes on WebAssembly’ga rakendus realiseeritud täielikult WebAssembly’s ning seega ei anna need ülevaadet selle kohta, kuidas WebAssembly ja JavaScript omavahel koos töötavad. Lisaks puudub hetkel efektiivne ning mugavalt kasutatav graafikute joonistamise teek suurte andmestikkude jaoks veebirakendustes.

Töö jaguneb kolmeks osaks: teoreetiline taust, realisatsioon ning võrdlusanalüüs. Töö teoreetilises osas antakse ülevaade WebAssembly’st ning selle eelistest ja puudustest. Lisaks tehakse kokkuvõtte eelnevatest uurimustest, mis on käesoleva tööga seotud. Järgmiseks kirjeldatakse näidisülesanne ning selgitatakse metoodikat võrdulusanalüüsi läbi viimiseks. Edasi seatakse nõuded loodavale teegile. Realisatsiooni osas kirjeldatakse graafikute teegi realisatsiooni ning arendusvoogu ning valideeritakse loodud teek nõutele. Võrdulsaanalüüsi osas viiakse läbi kiiruse võrdlus JavaScriptiga lahenduse ja WebAssembly’ga lahenduse vahel, tuuakse välja tulemused ning tehakse tulemuste põhjal järeldusi. Viimaks kirjeldatakse vajalikke edasiarendusi loodud teegile.

2 Teoreetiline taust ja varasemad uurimused

Selles peatükis annan ülevaate teoreetilisest taustast, mis on vajalik töö mõistmiseks ning lisaks teen kokkuvõtte varasematest uurimustest.

2.1 WebAssembly

WebAssembly on binaarne käsuformaat, mis võimaldab kompileerida kõrgkeeli (C, C++, Rust) veebirakendustesse. WebAssmebly on kasutatav nelja suurima veebilehitseja uuemates versioonides (Google Chrome - alates versioon 57, Microsoft Edge - alates versioon 16, Mozilla Firefox - alates versioon 52 ja Apple Safari - alates versioon 11 [1]). WebAssembly võimaldab veebis jooksutada eelkompileeritud koodi kasutamata pluginat ning avab seetõttu veebirakendustele uusi võimalusi. [2], [3]

Kuna WebAssembly puhul on tegemist suhteliselt uue (saadaval alates 2017. aasta märtsist) ja kiiresti areneva tehnoloogiaga, siis selles peatükis selgitan, mis WebAssembly on, kuidas see toimib ning millised on selle puudused. Lisaks annan ülevaate WebAssembly operatsioonikiirusest varasemate teadustööde põhjal.

2.1.1 Google NaCl ja Mozilla asm.js

Enne WebAssembly loomisega tegelema hakkamist tegelesid Google ja Mozilla oma tehnoloogiate loomisega, et brauserisse viia suuremat jõudlust nõudvaid rakendusi. Selles alapeatükis annan põgusa ülevaate nendest tehnoloogiatest ning selgitan, miks need tehnoloogiad edu ei saavutanud.

Google NaCl ja PnaCl

Google NaCl (*Native Client*) on liivakast (*sandbox*) eelkompileeritud C ja C++ programmikoodi efektiivseks ja turvaliseks jooksutamiseks veebibrauseris, sõltumata kasutaja operatsioonisüsteemist. NaCl on arhitektuurist sõltuv ning seega tekkis vajadus PNaCl (*Portable Native Client*) järele. PNaCl on arhitektuurist sõltumatu ning seetõttu võimaldab arendajatel kompileerida kirjutatud programmikoodi ühe korra, et jooksutada seda mistahes veebilehel, mistahes arhitektuuril. [4]

PNaCl oli lubav tehnologia jõudluse piirangute tõstmiseks veebirakendustes, kuid siiski ei saatnud seda edu. Google Chromel oli küll sisseehitatud PNaCl tugi, kuid teistel suurtel veebilehitsejatel mitte. Näiteks ei lisanud Mozilla PNaCl tuge selle puudliku API ja limiteeritud dokumentatsiooni pärast. Lisaks oli PNaCl rakendused "mustad kastid" ning neis esines potentsiaalseid turvariske. [3] Google Chrome plaanib PNaCl toe likvideerida 2019. aasta teises kvartalis (välja arvatud Chrome-i rakendused). [4]

Mozilla asm.js

Asm.js on range JavaScripti alamhulk, mida kompilaatorid saavad kasutada efektiivse sihtkeelena. [5] Asm.js on JavaScript, mis on limiteeritud nendele omadustele, millel on võimalik AOT (*ahead-of-time*) optimeerimine. AOT on brauserite JavaScripti mootorite poolt kasutatav tehnoloogia, mis võimaldab programmikoodi jooksutada efektiivsemalt kompileerides selle masinkoodiks. [3]

Nii Asm.js-i kui ka WebAssembly moodulid tuleb enne kompileermist laadida üle võrgu. Asm.js-i moodul on tekstifail, kuid WebAssembly moodul on binaarses formaadis, mis tähendab seda, et WebAssembly moodul on suuruselt väiksem ning seega on laadmisprotsess efektiivsem. [3] WebAssembly minimaalse elujõulise toote funktsionaalsus on laias laastus sama, mis on asm.js-i funktsionaalsus [6].

Üks põhjusi, miks asm.js asendus WebAssembly'ga on jõudlus. Nimelt oli asm.js limiteeritud funktsionaalsusele, mida võimaldab JavaScript, kuid WebAssembly ei ole selles osas limiteeritud ning võimaldab kasutada rohkemaid protsessori omadusi, näiteks 64-bitised täisarvud (operatsioonid nendega on kuni 4 korda kiiremad). [7]

2.1.2 Semantika

Selles peatükis selgitan lahti kolm olulist semantilist aspekti, mida on vaja mõista WebAssembly's arendamise puhul ning käesolevast tööst aru saamiseks.

Andmetüübid ja operaatorid

Hetkel on WebAssembly's saadaval 4 andmetüüpi: i32 (32-bitine täisarv), i64 (64-bitine täisarv), f32 (32-bitine ujukomaarv) ja f64 (64-bitine ujukomaarv). Igal parameetril ning lokaalsel muutujal on alati täpselt 1 andmetüüp. Funktsioonile saab kaasa anda 0 või rohkem parameetrit ning funktsioon tagastab 0 või rohkem väärtust. (WebAssembly

hetkeversioonis on maksimaalne funktsiooni poolt tagastatav väärtuste arv 1). [8], [9] Andmetüüpe juhitakse sisseehitatud operaatorite abil, mis peegeldavad kaasaegsetel protsessoritel kasutatavat masinkeelt (näiteks i32.add, f32.sub ja f64.ceil). [10]

Mälu

WebAssembly kasutab lineaarset mälu, mis on piiratud, bait-adresseeritav mäluruum. Mäluruumi suurus on WebAssembly puhul alati 64 KiB kordne. (1 KiB = 1024 B). [8] See on implementeeritud kasutades *TypedArray* andmetüüpi. [1] Lineaarset mälu võib vaadelda kui andmetüübita baitide massiivi. [11] Kuna WebAssembly'sse ei ole veel sisseehitatud prahikoristuse tuge, siis peavad rakendused ise enda poolt kasutatavat mälu haldama. [10]

Lineaarsele mälule ligi pääsemiseks on kasutusel *load* ja *store* operaatorid. Kõik *load* ja *store*"operaatorid kasutavad *little-endian* baidijärjestust väärtuste ja baitide vaheliseks teisendamiseks. Täisarvude laadimisel on võimalik spetsifitseerida nii kasutatava mälu suurus, mis on väiksem kui väärtuse andmetüübi suurus, kui ka märgistatus. [10]

Tüübid ja determinism

Kogu WebAssembly kood on tüübitud ning seda kontrollitakse enne kompileerimist. Kui esineb vigu, mida ei saa enne kompileerimist staatiliselt tuvastada, siis tekib lõks (*trap*). [10] Lõks tähendab seda, et WebAssembly koodi täitmine peatatakse ning antakse väliskeskkonnale teade ebareeglipärasest käitumisest. Kui tegemist on JavaScripti keskkonnaga (näiteks veebibrauser), siis lõksu tulemusena visatakse JavaScripti erand. [9] WebAssembly's ei esine määratlemata, kirjeldamata ega mittedeterministlikku käitumist. [10]

2.2 WebAssembly eelised

Kuna WebAssembly lisab veebirakenduste arendamisele keerukust, siis on küsimus, et miks peaks seda üldse kasutama. Selles peatükis annan ülevaate WebAssembly eelistest, millest peamine on operatsioonikiirus (ühe lihtsa võrdluse alusel on WebAssembly 10 korda kiirem kui JavaScript [12]) ning lisaks selgitan, miks on WebAssembly erinev Java apletitest ja Flashist.

2.2.1 WebAssembly kiiruse analüüs varasemate uurimuste põhjal

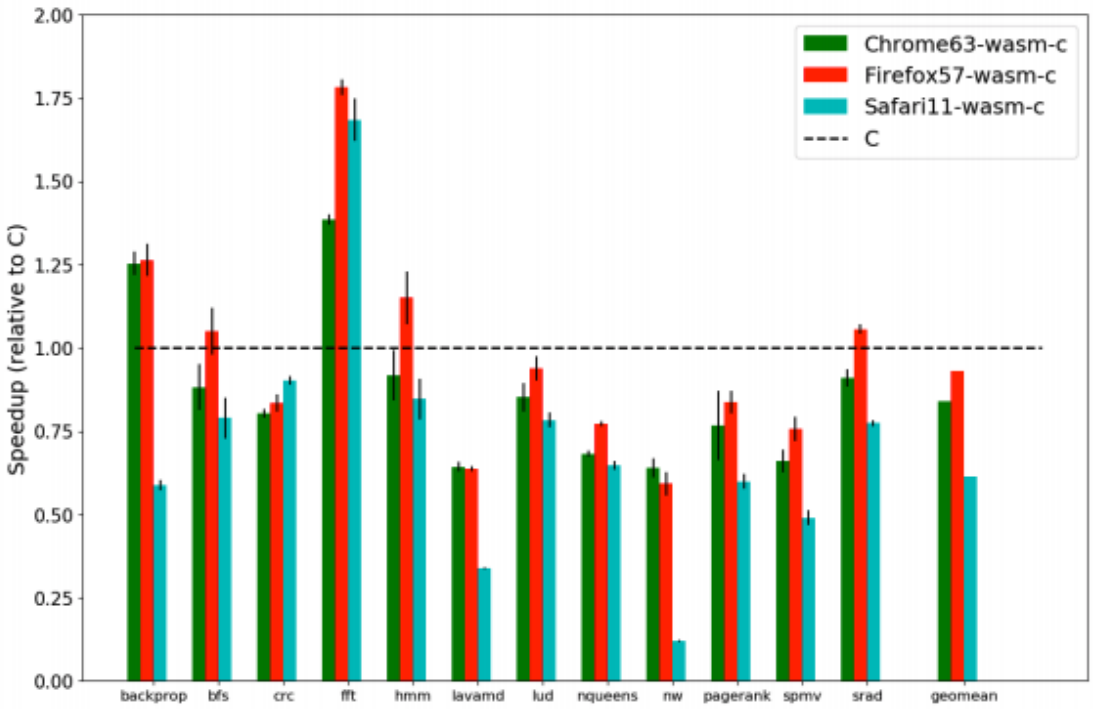
Webassembly eesmärk on kasutada ära tavalist riistvara võimekust erinevatel platvormidel, et jooksutada rakendust peaaegu sama kiiresti kui jooksutada selle sihtkeelt (näiteks C/C++) otse operatsioonisüsteemis [2]. WebAssembly eeliseks JavaScripti ees on selle suurem operatsioonikiirus [13]. WebAssembly kiirust võrreldes tavalise C jooksutamise kiirusega on võrreldud kolmes hiljutises uuringus. Samuti on uuritud WebAssembly kiirust võrreldes JavaScriptiga. Selles alapeatükis teen kokkuvõtte tulemustest. Siinkohal pean vajalikuks mainida, et kuna WebAssembly on kiiresti arenev tehnoloogia, siis antud tulemused ei pruugi enam 100% tõele vastata, kuid siiski annavad head ülevaate WebAssembly kiirusest.

Esimene analüüsitav uurimus viidi läbi 2018. aasta märtsist McGilli ülikooli Sable'i uurimisrühma poolt. [14] Uurimuses kasutati Ostrichi testikomplekti [15], mis on mõeldud arvutusmatemaatikas kasutatavate keelte jõudluse uurimiseks. Kasutatud Ostrichi võrdlusalus sisaldas 12 erinevat algoritmi, näiteks laiuti otsing, *nquuens* ja Google'i poolt loodud *page-rank*.

Töös võrreldi WebAssembly kiirust nii JavaScriptiga kui ka sama C koodiga, millest WebAssembly kompileeritud oli. Võrdlusanalüüs viidi läbi erinevatel seadmetel ning mitmes veebilehitsejas, millest Chrome (versioon 63.0.3239.84) ja Firefox (versioon 57.0.2) olid kasutatavad igal seadmel. Võrdlustulemuste esitamisel kasutati suhtelise jõudluse printsiipi, mille aluseks oli tavalise C koodi jooksutamise kiirus.

Webassembly vs C

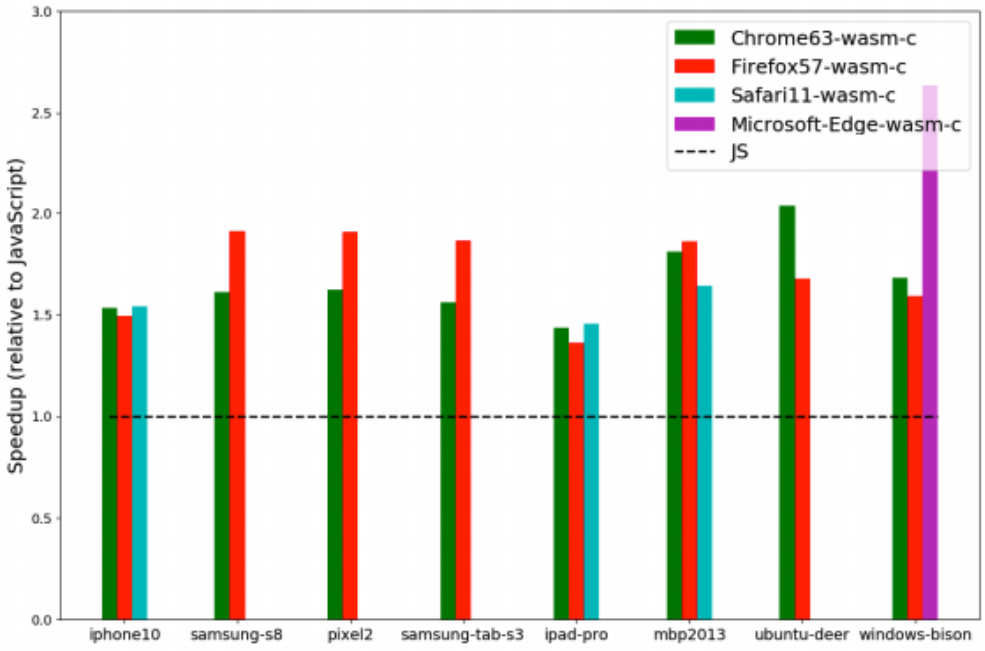
Uurimusest selgus, et WebAssembly kiirus veebilehitsejas on vähemalt 75% (välja arvatud Safari11 Macbookil) tavalisest C jooksutamise kiirusest, kusjuures koguni 5 algoritmi puhul oli WebAssembly kiirem kui tavaline C, kuid C keskmine kiirus oli siiski parem kui WebAssembly oma. Kõige parem tulemuse WebAssembly'ga andis Firefox, mille jõudlus oli kõige lähemal C jõudlusele. Ubuntut kasutaval arvutil oli Firefoxis WebAssembly jõudlus koguni parem kui tavalise C jõudlus. WebAssembly ja C võrdlusanalüüsi tulemused on esitatud joonisel 1.



Joonis 1. WebAssembly jõudlus võrreldes C-ga MacBook Pro 2013 peal. [14]

WebAssembly vs JavaScript

Uurimusest selgus, et WebAssembly jõudlus edestab selgelt JavaScripti oma ning seega on kindlasti kasulik WebAssembly’t kasutada arvutusmatemaatika probleemide lahendamiseks. Androidi kasutatavate seadmete peal oli WebAssembly 2 korda kiirem kui JavaScript ning Microsoft Edge puhul oli vahe 2.5-kordne. WebAssembly ja JavaScripti võrdlusanalüüsi tulemused on esitatud joonisel 2.



Joonis 2. WebAssembly jõudlus võrreldes JavaScriptiga erinevatel platvormidel. [14]

Teine analüüsitav uurimus viidi läbi 2019. aasta jaanuaris Massachusetts Amherst ülikooli teadlaste poolt. [16] Selles uurimuses loodi BROWSIX-ile [17] laiendus BROWSIX-WASM, mis võimaldas esimest korda jooksutada muutmata kujul WebAssembly'sse kompileeritud Unixi aplikatsioone otse veebibrauseris. Uurimus erineb eelnevast selle pooles, et testitav rakendus on võrreldes Ostrichi testikomplekti algoritmidega tunduvalt suurem.

Uurimusest selgus, et WebAssembly on keskmiselt 1.3 korda kiirem kui JavaScript, kuid tavalise C ja WebAssembly vahe oli suhteliselt suur: FireFoxis oli WebAssembly 1.5 korda aeglasem ja Chrome puhul 1.9 korda aeglasem. Siit võib järeldada, et suurte rakenduste puhul ei ole WebAssembly jõudnud veel oma eesmärkideni.

Kolmas uurimus viidi läbi 2018. aasta veebruaris. [18] Uurimuse üks eesmärkidest oli võrrelda C koodi kiirust nii JavaScriptiga kui ka Emscripten SDK abil kompileeritud ASM.js-i ja WebAssembly'ga. Uurimuses testiti kõiki tehnoloogiaid nii lihtsate algoritmidega (näitkes 1 miljoni suuruse massiivi täitmine suvaliste numbritega, 10 miljoni paari suvalise täisarvu võrdlemine), kui ka keerukamate algoritmidega (Floyd-Warshall, Huffman Coding). Kokku viidi võrdlusanalüüs läbi kasutades 10 algoritmi. Uurimuse tulemusena selgus, et WebAssembly oli iga algoritmi puhul kiirem kui JavaScript ning taaskord suutis WebAssembly mõne algoritmi puhul (näiteks massiivi

täitmine, täisarvude paaride võrdlus) olla kiirem kui tavaline C kood. Võrreldes ASM.js-iga oli WebAssembly 8 algoritmi puhul kiirem ning 2 algoritmi puhul täpselt sama kiire.

Võrdlusanalüüsid C keelega sisendab optimismi veebibrauserites kasutatavate rakenduste peaaegu sama kiireks toimimiseks, nagu otse operatsioonisüsteemis kasutatavad rakendused toimivad. Kindlasti peab see veel arenema, eriti suuremates rakendustes kasutamiseks. Võrdlusanalüüsid JavaScriptiga näitavad selgelt, et WebAssembly on kiirem kui JavaScript ning kolmandas analüüsitavast uurimusest selgus, et asm.js-i asendamine WebAssembly'ga on samuti õigustatud.

2.2.2 WebAssembly keeled

Teine WebAssembly eelis on võimalus kaasata veebirakenduste loomisesse arendajaid, kes on tugevad mõnes muus keeles peale JavaScripti. Hetkel on võimalik WebAssembly'sse kompileerida üle 20 erineva keele, näiteks Rust, C/C++, C#/.Net, Java, Go ja Python. [19]

2.2.3 WebAssembly vs Java apletid ja Flash

Kui öelda, et WebAssembly võimaldab kasutada kõrgkeeli nagu C/C++ ja Rust veebirakenduste loomisel, siis võib tekkida küsimus, kas Java apletid või Flash on tagasi. Selles jaotises selgitan, mis on WebAssembly erinevus ülal mainitud tehnoloogiatest.

Kõige suurem erinevus WebAssembly ja ülal mainitud tehnoloogiate vahel on see, et WebAssembly ei vaja töötamiseks pluginat, vaid on osa veebiplatvormist. See tähendab, et näiteks Java apletid jooksid nii-öelda kastis ning suhtlus JavaScriptiga puudus. [20] WebAssembly eelis ongi see, et see võimaldab efektiivsust suhtlust JavaScriptiga. Näiteks kui luua funktsioon WebAssembly's, siis on seda võimalik JavaScriptis kutsuda ning samuti vastupidi.

2.3 WebAssembly puudused

Webassembly esmane versioon, mis avalikustati 2017. aasta märtsis [21] on kõigest minimaalne elujõuline toode, mida arendatakse pidevalt edasi [22]. Et Webassembly't efektiivsemaks saada ning võimaldada suuremate koodimahtudega rakendusi (näiteks Adobe Photoshop või AutoCad) tuleb üle saada mitmetest puudustest, mille

lahendamisega hetkel aktiivselt töötatakse. Selles peatükis kirjeldan hetkel esinevaid puudusi.

Mitmelõimelisus

Webassembly ei toeta veel mitmelõimelisust. Mitmelõimelisuse lisamine kallal käib hetkel aktiivne töö. [8] Kuna riistvara on pidevas arengus ning protsessorite võimekus ja tuumade arv suureneb, siis mitmelõimelisuse puudumine ei võimalda riistvara poolt pakutavaid ressursse efektiivselt ära kasutada. Et kasutada mitut tuuma korraga on vaja mitmelõimelisuse toetust [11].

SIMD

Peale mitmelõimelisuse on veel üks tehnoloogia, SIMD (*Single instruction, multiple data*), mida Webassembly hetkel ei võimalda. [11] SIMD on tehnoloogia, mis võimaldab üheaegselt töödelda mitut *scalar*-tüüpi muutuja. SIMD puudumine tähendab taas seda, et riistavara pakutavat võimekust ei suuda Webassembly hetkel täielikult ära kasutada.

Mäluaadressid

Hetkel kasutab Webassembly 32-bitiseid mäluaadresse, mis tähendab seda, et maksimaalne mälukasutatavus Webassembly rakenduse poolt on 4 GB, mida on liiga vähe [11]. Näiteks on Adobe Photoshopi soovitatav vahemälu vähemalt 8 GB [23] ning seega sellisel kujul seda WebAssembly's jooksma panna ei saaks. Seega peab WebAssembly kindlasti üle minema 64-bitistele mäluaadressidele, mis võimaldab kasutada 16 EB (10^9 GB) mälu. Hetkel 32-bitiste mäluaadresside puhul limiteerib tarkvara riistvara, kuid 64-bitiste puhul limiteerib riistvara tarkvara võimalusi.

Kompileerimisprotsess

Veebirakenduste puhul ei ole oluline ainult töötamiskiirus, vaid ka laadimiskiirus. Hetkel töötab WebAssembly kompileerimine nii, et kõigepealt laaditakse fail alla ning, siis hakatakse seda kompileerima, mis ei ole optimaalne lahendus.

Kompileermisprotsessi optimeerimiseks töötatakse hetkel kahe lahenduse kallal. Esimese lahenduse puhul kompileeritakse WebAssembly faili samal ajal kui seda laetakse. Hetke seisuga on kompileerimiskiirus Firefoxi puhul kiirem kui faili laadimiskiirus, mis

tähendab seda, et kui fail on allalaaditud on see sisuliselt ka kompileeritud. Teiste veebilehitsejate puhul lahenduse kallal töötatakse. [11]

Teine lahendus, mida Firefoxis juba kasutatakse on astmeline kompilaator. Kasutatakse kahte kompilaatorit, milles esimene kompileerib WebAssembly koodi laadimisajal, kuid mille väljund ei ole optimeeritud. Teine kompilaator hakkab tööle, kui fail on laaditud. Teise kompilaatori väljund on optimeeritud ning kui see on oma töö lõpetanud vahetatakse esimene kompileeritud versioon teise vastu. Antud lahendus võimaldab parandada laadimiskiirust ning samuti pärast teise kompilaatori töö lõpetamist on töökiirus optimeeritud. [11]

HTTP Vahemälusse salvestamine

Kui anda WebAssembly kompilaatorile sama sisend, siis väljund on alati sama, mis tähendab seda, et laadimiskiirust parandada oleks otstarbekas salvestada kompileeritud masinkood HTTP vahemällu. Kui navigeerida samale lehele kaks korda, siis esimene kord laaditakse alla WebAssembly fail ning kompileeritakse see, teisel külastusel aga võetakse masinkood otse HTTP vahemälust ning kompileerimist enam vaja ei ole. [11]

Andmetüübid

Enamikul WebAssembly kasutusjuhtudest on vaja saata sellele sisendandmed ning WebAssembly tagastab väljundandmed. Antud probleem on hetkel keerukam, kui see esmapilgul tundub. Nimelt hetkel saab WebAssembly aru ainult numbritest, mis tähendab, et kui on vaja parameetrina ette anda mõni sõne või objekt, siis tuleb see konverteerida numbriteks ning paigutada need lineaarsesse mälusse. Seejärel tuleb anda WebAssembly'le lineaarse mälu asukoht. [11] Selline komponentide vaheline suhtlus on ebamugav ning konverteerimine on lisa ajakulu.

Prahikoristus

Hetkel toetab WebAssembly ainult lineaarset mälu, mis sobib laitmatult näiteks programmeerimiskeelte C, C++ ja Rust jaoks. Enamik laialt kasutatavaid programmeerimiskeeli vajavad töötamiseks prahikoristust. Praegu ainus viis, kuidas neid keeli kasutada on implementeerida prahikoristus WebAssembly's kohandatud koodiga. Et WebAssembly lõplikud eesmärgid täita peab see olema võimeline:

- viitama DOM-i ja veebi API objektidele otse Webassembly koodist;
- kasutama veebi API-t otse, mitte läbi JavaScripti
- efektiivselt eraldama ja manipuleerima prahikoristuse objekte otse WebAssembly koodist. [24]

2.4 WebAssembly arendusvoog

Selles peatükis kirjeldan, kuidas arendada WebAssembly mooduleid, võttes aluseks programmeerimiskeele C. Näitena loon lihtsa C funktsiooni ning kompileerin sellest WebAssembly koodi kasutades webassembly.studio-t [25]. Antud näite demostreerimiseks ei ole kompilaator oluline ning võiks kasutada ka mõnda teist varianti, näiteks emscripten SDK [26], mis on laialt kasutatav tööriist C ja C++ koodi kompileerimiseks WebAssembly'sse. Lisaks kirjeldan .wat ja .wasm failide olemust.

C kood

Lihtsa näitena loon ühe funtsiooni, mis võtab sisendiks kaks täisarvu ning tagastab nende summa. Funktsioon näeb välja selline:

```
#define WASM_EXPORT __attribute__((visibility("default")))

WASM_EXPORT
int addTwoIntegers(int a, int b) {
 return a + b;
}
```

Joonis 3. Näide programmeerimiskeel C funktsioonist, mida on võimalik WebAssembly'sse kompileerida

Programmikoodi esimesel real defineerin makro WASM_EXPORT ning sean selle atribuudi *visibility* väärtuseks *default*. See ütleb kompilaatorile, et kõik WASM_EXPORT-iga annoteeritud funktsioonid tuleb eksportida, mis võimaldab neid JavaScriptis kutsuda. Funktsioon ise võtab argumendina 2 täisarvu ning tagastab nende summa. Nüüd järgmiseks sammuks on kirjutatud programmikoodi kompileerimine WebAssembly'sse.

WebAssembly tekstiline formaat (.wat)

WebAssembly'l on 2 esitusviisi: tekstiline formaat (.wat) ning binaarne formaat (.wasm). Tekstiline formaat on inimesele loetav ning seda on võimalik ka ise kirjutada. Ülal toodud C programmikoodi kompileerimisel WebAssembly'sse saame järgneva teskstilise esituse:

```
(module
 (type $t0 (func))
 (type $t1 (func (param i32 i32) (result i32)))
 (func $__wasm_call_ctors (type $t0))
 (func $addTwoIntegers (export "addTwoIntegers") (type $t1) (param $p0
i32) (param $p1 i32) (result i32)
   get_local $p1
   get_local $p0
   i32.add)
 (table $T0 1 1 anyfunc)
 (memory $memory (export "memory") 2)
 (global $g0 (mut i32) (i32.const 66560))
 (global $__heap_base (export "__heap_base") i32 (i32.const 66560))
 (global $__data_end (export "__data_end") i32 (i32.const 1024)))
```

Joonis 4. Näide C programmikoodist kompileeritud WebAssembly tekstiline esitus.

Meie jaoks olulised kohad on märgistatud rasvaselt. Kõige pealt defineeritakse tüüp „$t1", mis on funktsioon, mille sisendiks on 2 i32 tüüpi muutujat ning samuti tagastatavaks väärtuseks on i32 tüüpi väärtus. Edasi näeme, et defineeritakse funktsioon „$addTwoIntegers", mis eksporditakse nimega „addTwoIntegers". Defineeritud funktsioon on „$t1" tüüpi ning võtab 2 parameetrit „$p0" ja „$p1", mis on mõlemad i32 tüüpi ja tagastab i32 väärtuse. Funktsioon võtab lokaalsed muutujad „$p0" ja „$p1" ning kasutades i32 operaatorit add liidab need kokku.

WebAssembly binaarne formaat (.wasm)

WebAssembly teine esitusviis on binaarne ning inimese jaoks loetamatu. Kompileerides ülal toodud C programmikoodi WebAssembly'sse saame binaarse esituse. Kuna binaarne esitus on võrdlemisi mahukas, siis on see toodud lisas 1. Antud binaarne kood vastab ülaltood WebAssembly teksilisele esitusele (.wat).

WebAssembly mooduli laadimine JavaScriptis

Et loodud WebAssembly moodulit veebirakenduses kasutada on vaja see JavaScriptis laadida, kompileerida ning algväärtustada. Loodud WebAssembly mooduli kasutamiseks JavaScriptis on vajalik järgnev:

```
fetch('../out/main.wasm').then(response =>
 response.arrayBuffer()
).then(bytes => WebAssembly.instantiate(bytes)).then(results => {
 instance = results.instance;
 console.log(instance.exports.addTwoIntegers(5, 7));
}).catch(console.error);
```

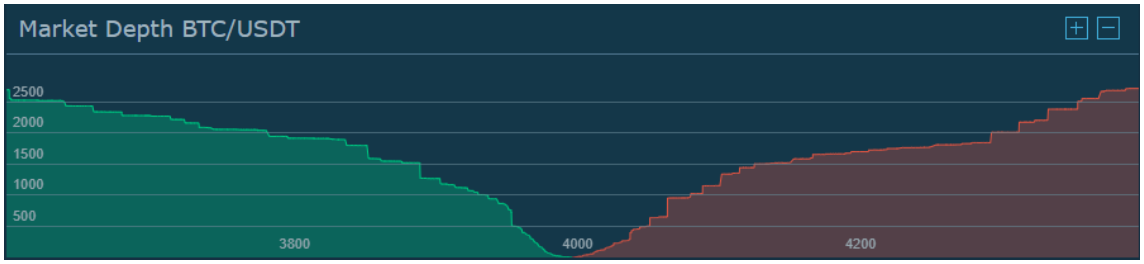
Joonis 5. WebAssembly mooduli laadimine, kompileerimine ja algväärtustamine JavaScriptis.

Kogu faili laadimisprotsess on ülesehitatud kasutades *Promise* objekte. Kõige pealt laaditakse WebAssembly fail kasutades *fetch* API-t. Edasi laaditakse binaarne kodeering *arrayBuffer* objekti. Saadud objekt antakse „WebAssembly.instantiate“ funktsioonile, mis kompileerib WebAssembly mooduli ning tagastab objekti, mis sisaldab „WebAssembly.Module“ ning „WebAssembly.Instance“ objekte. Nüüd on võimalik eksporditud funktsioone JavaScriptis kasutada. Eksporditud funktsioonid asuvad „WebAssembly.Instance.exports“ objektis.

3 Näidisülesanne

Näidisprobleemina võib tuua olukorra, kus on tarvis visualiseerida krüptoturgude tellimusraamat (elektrooniline nimekiri kõigist ostu- ja müügitellimustest konkreetse kauplemispaari jaoks [27]). Sellisel juhul arvutatakse graafikupunktid tavaliselt serverirakenduses, kuid alati ei ole see võimalik. Näiteks võivad andmed tulla veebirakendusele erinevate vahendajate veebisoklitest [28]. Sellisel juhul on vaja need andmed kombineerida ning graafikupunktid arvutada brauserirakenduses.

Arvutusi võimaldab WebAssembly teha kiiremini kui JavaScript ning seetõttu on WebAssembly selles olukorras kasulik. Selle näite realiseerimine on kasulik, sest JavaScriptile ei ole konkreetselt kirjeldatud probleemile keskendunud teeki. On küll mõni teek, mis seda võimaldab, kuid need osutvad ebaefektiivseks suurte andmestikkude korral või on kirjeldatud graafiku loomiseks vajalik koodihulk liiga mahukas. Näiteks võttis AmChartsi [29] pakutava graafiku loomine aega 5.1 sekundit (nii ostu- kui ka müügitellimusi oli 500). Teise näitena võib tuua D3.js [30] teegi peale ehitatud graafiku [31]. Näide tellimusraamatu sügavuse graafikust on toodud joonisel 6.



Joonis 6. Näide BTC ja USDT kauplemispaari visualiseeritud tellimusraamatust HitBTC vahendusplatvormis [32].

Üks samm kirjeldatud graafikust edasi on graafik, kus kuvatakse mitme kauplemispaari tellimusraamatut korraga. Selle graafiku eelduseks on, et kauplemispaaride baasvaluuta on sama (krüptoturgudel näiteks paarid BTCUSD ja ETHUSD). Graafiku vertikaalteljel kuvatakse tellimuste summa suurus baasvaluutas ning horisontaalteljel hinna protsentuaalne hälve hetke hinnast. Nii saab kõik kauplemispaarid summeerida ning need graafikul kuvada. Graafik annab ülevaate üldisest turu liikumisest (näiteks, kui

baasvaluuta on BTC, siis graafikult saab välja lugeda, kas hetkel tahetakse valuutat BTC pigem müüa või osta ning analüütikud saavad selle põhjal järeldusi teha). Selline graafikutüüp on selle töö raames plaanis luua. Töö käigus nimetan graafikutüüpi tellimusraamatu suhtelise sügavuse graafikuks.

4 Võrdlusanalüüsi metoodika

Selles peatükis selgitan metoodikat WebAssembly ja JavaScripti võrldlusanalüüsi läbi viimiseks. Toon välja, millistel seadmetel ning millistes veebibrauserites võrdlusanalüüs läbi viiakse, lisaks annan ülevaate valitud andmestikkude kohta. Et võrdlusanalüüs läbi viia, siis kogu WebAssembly's arendatud funktsionaalsus tuleb dubleerida JavaScriptis.

Võrdlusanalüüsi eesmärgiks ei ole mitte WebAssembly ja JavaScripti kiiruste võrdlemine, vaid WebAssembly'ga ja puhta JavaScriptiga lahenduse kiiruse võrdlemine. See tähendab, et kiiruste võrdlemisel, on sisse jäetud ka need JavaScripti funktsioonid, mis on vajalikud graafiku analüüsitava funktsionaalsuse saavutamiseks. Võrdlusanalüüs peaks andma vastuse küsimustele:

- Kas ja kui palju kiiremini toimib WebAssembly'ga lahendus võrreldes JavaScriptiga lahendusega?
- Kuidas on seotud graafikul kuvata andmestiku suurus ja WebAssembly'ga lahenduse ning JavaScriptiga lahenduse kiiruse vahe?
- Mis suurustega andmestikkude korral on otstarbekas WebAssembly kasutusele võtta graafikute loomise juures?

4.1 Kasutatavad seadmed

Et saada piisav ülevaade WebAssembly ja JavaScripti jõudluse erinevusest viin võrdlusanalüüsi läbi erinevatel seadmetel. Võrdlusanalüüsiks õnnestus saada 3 erinevat nutiseadet, millest 1 on nutitelefon iPhone 7 Plus. Arvutitena on kasutusel Windowsi kasutav lauaarvuti ning macOS operatsiooni süsteemi kasutav MacBook Pro 2017. Kõik kasutusel olevad seadmed kasutavad erinevat operatsioonisüsteemi. Kasutatavate seadmete spetsifikatsioon on välja toodud tabelis 1.

Tabel 1. Võrdlusanalüüsis kasutatavate seadmete spetsifikatsioon.

Nutiseade	**Protsessor**	**Muutmälu**	**Operatsioonisüsteem**
iPhone 7 Plus	Apple A10 Fusion Quad-core @2.34 GHz	3 GB	iOS 12.2
Windowsiga lauaarvuti	Intel(R) Core(TM) i5-6600K CPU @ 3.5 GHz	32 GB	Windows 10 Pro versioon 1803
MacBook Pro 2017	Intel Core i5 @ 2.3 GHz	8 GB	macOs Mojave versioon 10.14.4

4.2 Kasutatavad veebibrauserid

Võrdlusanalüüsi läbi viimisel kasutan töös 4 erinevat veebibrauserit: Google Chrome'i, Mozilla Firefoxi, Apple Safarit ja äsja ilmunud Microsoft Chromium Edge'i, mis on küll arendusjärgus, kuid testimiseks kättesaadav. Leian, et 4 erinevat brauserit on piisav järelduste tegemiseks WebAssembly ja JavaScripti jõudluse analüüsil. Võrlusanalüüsis kasutatud veebibrauserite versioonid on esitatud tabelis 2. Lisaks on välja toodud veebibrauserite versioonid, mida konkreetsetel seadmetel kasutati.

Tabel 2. Võrdlusanalüüsiks kasutatavate veebibrauserite detailandmed.

Veebibrauser	**Windowsiga arvuti**	**MacBook Pro**	**iPhone 7 Plus**
Google Chrome	73.0.3683.103	73.0.3683.103	73.0.3683.68
Mozilla Firefox	66.0.3	66.0.3	16.0.14732
Apple Safari	-	12.1	12.2
Microsft Chromium Edge	74.1.96.24 dev	-	-

4.3 Võrdlusanalüüsiks kasutatavad andmestikud

Kuna töö käigus loon 2 erinevat graafikutüüpi, siis andmestikud jagunevad samuti kaheks tüübiks. Võrdlusanalüüsi viin läbi erineva suurusega andmestikkude korral, et saada aimu, mis suurusega andmestikkude korral on kasulik üldse WebAssembly't kasutada,

arvestades, et iga uue tehnoloogia kasutusele võtmisel on õppimiskõver ning rakenduse keerukus suureneb. Kuna mõlema graafikutüübi puhul on oluline nii graafiku joonistamise kui ka andmete sisestamise/uuendamise osa, siis võrdlen mõlema funktsionaalsuse kiirusi.

4.3.1 Joondiagrammi testandmed

Joondiagrammi testandmeteks võtan krüptovaluutade vahendusplatvormi Binance erinevate kauplemispaaride hinnad minutiliste intervallidena vahemikus 01.11.2017 – 31.03.2018. Testiandmed on kättesaadavad https://www.kaggle.com/binance/binance-crypto-klines.

Kuna andmed on *csv* formaadis, siis kõige pealt töötlen andmed ümber *json* formaati, et neid JavaScriptis lihtsam hallata oleks. Edasi võtan andmetest välja kindla suurusega alamhulga ning viin läbi võrdlusanalüüsi JavaScripti ja WebAssembly vahel. Kui mingi kauplemispaari kohta on piisavalt andmeid et andmehulga suuruse kriteerium täita, siis kasutan võrdlusanalüüsis ühe kauplemispaari andmeid, kui aga ühe kauplemispaari kohta piisavalt andmeid ei ole, siis kasutan mitme kauplemispaari andmeid (mitu joondiagrammi ühel graafikul). Võrdlusanalüüsis kasutatavad andmestikkude suurused on esitatud tabelis 3.

Tabel 3. Joondiagrammi võrdlusanalüüsis kasutatavate andmestikkude suurused.

Võrdlusanalüüsi iteratsiooni identifikaator	**Andmepunktide arv**
line_1	100
line_2	1000
line_3	10000
line_4	100000
line_5	500000

4.3.2 Tellimusraamatu suhtelise sügavuse graafiku testandmed

Tellimusraamatu suhtelise sügavuse graafiku testandmed võtan otse krüptovaluutade vahendusplatvormi hitBtc veebisoklist. Võrdlusanalüüsiks kasutan töös kõikide kauplemispaaride, mille baasvaluuta on Ameerika dollar (USD), tellimusraamatute andmeid. Selle graafikutüübi puhul viin võrdlusanalüüsi läbi sama suure hulga andmepunktide peal, kuid maksimaalne hinna protsentuaalne hälve hetke hinnast

varieerub. See tähendab, seda andmeid on iga võrdluse puhul sama palju, kuid kõiki neid ei käida graafiku joonistamisel läbi. Kuna andmepunktide arv on ajas muutuv suurus, siis ei ole võimalik täpset arvu öelda, kuid erinevatel ajahetkedel on andmepunktide arv jäänud vahemikku 90000 – 100000. Võrdlusanalüüsis kasutatavad hinna protsentuaalse hälbe suurused on esitatud tabelis 4.

Tabel 4. Tellimusraamatu suhtelise sügavuse graafiku võrdlusaanalüüsis kasutatavad hinna hälbed.

Võrdlusanalüüsi iteratsiooni identifikaator	**Joonistamisse kaasatud andmepunktide arv**	**Hinna protsentuaalne hälve**
depth_1	6000 - 6500	3%
depth _2	14000 - 15000	10%
depth _3	25000 - 26000	25%
depth _4	36000 - 38000	50%
depth _5	50000 - 52000	99%

4.4 Mõõtmised

Kõik töös esitatud mõõtmistulemused on 1000 mõõtmise summa aritmeetilised keskmised. Mõõtmistulemused on esitatud JavaScriptiga lahenduse kiiruse suhtena WebAssembly'ga lahenduse kiirusesse. Selline suhe näitab mitu korda kiirem on WebAssembly'ga lahendus JavaScriptiga lahendusest.

Kuna selles töös on oluline küsimus, kui palju võimaldab WebAssembly rakenduste toimimise kiirust suurendada, mitte konkreetselt WebAssembly ja JavaScripti identse programmikoodi kiiruse vahe, siis on mõõtmistesse sisse jäetud ka need JavaScripti funktsioonid, mis on graafiku joonistamiseks vajalikud. Teisisõnu kui mõõdetakse graafiku joonistamise kiirust, siis see hõlmab endast nii koordinaatite arvutamist kui ka SVG elementide lisamist HTML-i. Seega on mõõtmistulemustesse sisse arvestatud WebAssembly ja JavaScripti omavaheliseks suhtluseks kulunud aeg.

5 Nõuded teegile

Selles peatükis esitan peamised nõuded loodavale graafikute teegile. Nõuded jagunevad kolme kategooriasse: nõuded, mis kehtivad kõikide graafikutüüpide kohta (esitatud tabelis 5), nõuded joondagrammile (esitatud tabelis 6) ja nõuded tellimusraamatu suhtelise sügavuse graafikule (esitatud tabelis 7).

Tabel 5. Kõikide graafikutüüpide kohta kehtivad nõuded.

Identifikaator	**Kirjeldus**
fn_1	Graafiku kogusuurus (kõrgus ja laius) peab olema võrdne selle konteineri suurusega.
fn_2	Graafikul peab olema visuaalselt vaadates arusaadav ning mõistlik joonestik, mis arvutatakse vastavalt graafiku suurusele (*grid*).
fn_3	Graafiku teljestikul peavad olema kuvatud tekstilised väärtused. Iga väärtus peab asuma vastava joonestiku joone keskel.
fn_4	Graafik peab olema veebibrauseri akna suurusega kohanduv (*responsive*).
fn_5	Graafiku arvutusosad peavad olema realiseeritud nii WebAssembly's kui ka JavaScriptis, et saaks läbi viia võrdlusanalüüsi.
fn_6	Graafiku andmed peavad olema uuendatavad.
fn_7	Graafikul peab olema kursor, mis näitab kasutaja hiire kursori asukohta graafiku teljestike suhtes.
fn_8	Graafikul hiire kursoriga liikudes, peab kuvama info selles punktis olevate väärtuste kohta. Lisaks peab kuvama ka antud punktis teljestiku väärtust.
mfn_1	Kogu programmikood peab olema kommenteeritud ning vastama „puhta koodi" põhimõtele.
mfn_2	Graafikute loomise teek peab olema dokumenteeritud ning arendajatele arusaadavalt kasutatav.

Tabel 6. Joondiagrammi nõuded.

Identifikaator	**Kirjeldus**
fn_9	Graafik peab toetama mitme joone lisamist.
fn_10	Iga joone stiil (värv, paksus jne) peab olema valitav.

Tabel 7. Tellimusraamatu suhtelise sügavuse graafiku nõuded.

Identifikaator	**Kirjeldus**
fn_11	Graafik peab toetama mitme kauplemispaari tellimusraamatu andmeid.
fn_12	Ostutellimused peavad graafikul olema esitatud rohelisena ning müügitellimused punasena.
fn_13	Kui hiire kursor paikneb graafikul, siis graafikut uuesti ei tohi joonistata (Et kasutajal oleks võimalik konkreetsel ajahetkel järeldusi teha).

6 Realisatsioon

Selles peatükis kirjeldan teegi loomise arendusprotsessi ning annan ülevaate loodud teegist. Realisatsiooni lähtekoodi hoidla link on esitatud lisas 4.

6.1 Arenduskeskkonna seadistamine

Selles jaotises selgitan töös kasutatud arenduskeskkonda ning põhjendan, miks kasutasin valitud tööriistu, lisaks toon välja valitud arenduskeskkonna puudused ning keskkonna seadistamise sammud.

6.1.1 IDE ja kasulikud laiendused

Redaktorina kasutasin töös IntelliJ poolt loodud WebStormi, mille saab alla laadida: https://www.jetbrains.com/webstorm/?fromMenu. Enamik arenduseks vajalikke laiendusi on juba redaktori paigaldamisel olemas. Lisaks paigaldasin node.js toe (NodeJS) ja C++ toe (cppcheck). Kuigi WebStorm on loodud veebirakenduste loomiseks, ei ole see veel WebAssembly'ga töötamiseks optimeeritud. Arenduse käigus tulid välja järgmised puudused:

- puudub programmeerimiskeele C++ täielik tugi – on olemas küll laiendus (cppcheck), mis lisab süntaksi esitletõstmise, kuid puuduvad elementaarsed soovitused sisseehitatud funktsioonidele;
- puudub .wasm failide vaatamise tugi ning seega on kompileeritud WebAssembly kood „must kast". Kuigi WebAssembly rakenduse loomisel, ei ole otseselt vajalik .wasm faili sisu inspekteerida, siis esmasel katsetamisel annab see kindlasti parema arusaamise WebAssembly'st;
- puudub WebAssembly ametliku kodulehe poolt soovitatud C++ kompilaatorispetsiifilise (Emscripten) programmikoodi tugi.

Kui alustada arendamist uuesti nullist, siis valiksin kasutamiseks teise IDE, näiteks on Microsoft Visual Studiol esimese kahe puuduse lahendamiseks laiendused olemas ning on seega ka mugavam lahendus arendamiseks. Kuna Visual Studio võimekuse avastasin,

kui töö oli juba arendusjärgus, siis ei olnud otstarbekas selle kasutamisele üle minna, kuna keskkonna seadistamine ja tundma õppimine võtab aega.

6.1.2 C++ kompilaator WebAssembly'sse

C++ koodi kompileerimiseks WebAssembly'sse kasutasin töös Emscripten SDK-d, mis on kõige laiemalt levinud ning samuti WebAssembly ametliku kodulehe poolt soovitatud. Emscripten SDK seadistamiseks on vajalikud järgnevad sammud [26]:

- git clone https://github.com/emscripten-core/emsdk.git
- cd emsdk
- ./emsdk install latest (Windowsi operatsioonisüsteemis emsdk install latest)
- ./emsdk activate latest (Windowsi operatsioonisüsteemis emsdk activate latest)
- source ./emsdk_env.sh (Windowsi operatsioonisüsteemis emsdk_env.bat)

C++ programmikoodi kompileerimiseks WebAssembly'sse tuleb anda kompilaatorile ette lähtefaili asukoht, väljundfaili asukoht ning lipp „-s WASM=1", mis ütleb kompilaatorile, et kompileeritakse WebAssembly failiks [33]. Ilma mainitud lippu kasutamata, kompileeritakse sisend asm.js-iks.

Näiteks main.cpp faili kompileerimiseks WebAssembly failiks on vajalik järgnev sisend: „em++ main.cpp –s WASM=1 – o main.js", kus „main.cpp" on lähtefail ning „main.js" väljundfail. Käsu tulemusena loob kompilaator kaks faili: main.wasm ja main.js. Main.wasm on WebAssembly fail ning main.js on JavaScripti fail, mis laeb ja algväärtustab WebAssembly koodi ning sisaldab funktsionaalsust, mis võimaldab suhtlust JavaScripti ja WebAssembly koodi vahel.

6.1.3 Arendusserver

Arendusserveri ülesseadmiseks on kasutusel node.js (alla laaditav: https://nodejs.org/en/) ning selle raamistik express.js. Paketihaldurina kasutasin npm-i. Arendusserveri seadsin üles järgnevalt:

- lõin kataloogi masterThesis („mkdir masterThesis") ning navigeersin sinna („cd masterThesis");

- sisestasin käsu „npm init“ ning valisin kõik vaikimisiväärtused;
- installeerisin express.js-i („npm install express“)
- lõin failid index.html ja index.js, index.js faili lõin expressi serveri, mis laeb faili index.html

6.2 Loodud rakenduse arhitektuur

Selle alapeatükis selgitan kahe loodud graafikutüübi (ajaseeriatel põhinev joondiagramm ja tellimusraamatu suhtelise sügavuse graafik) arhitektuuri – selgitan, millised osad toimuvad JavaScriptis ja millised WebAssembly's.

Mõlemad graafikutüübid koosnevad laiaslaastus kahest osast: JavaScripti klass ning samanimeline WebAssembly klass, mis on klassi „Chart“ alamklass. Kogu kommunikatsioon kasutaja ja WebAssembly vahel käib kasutades JavaScripti klassi.

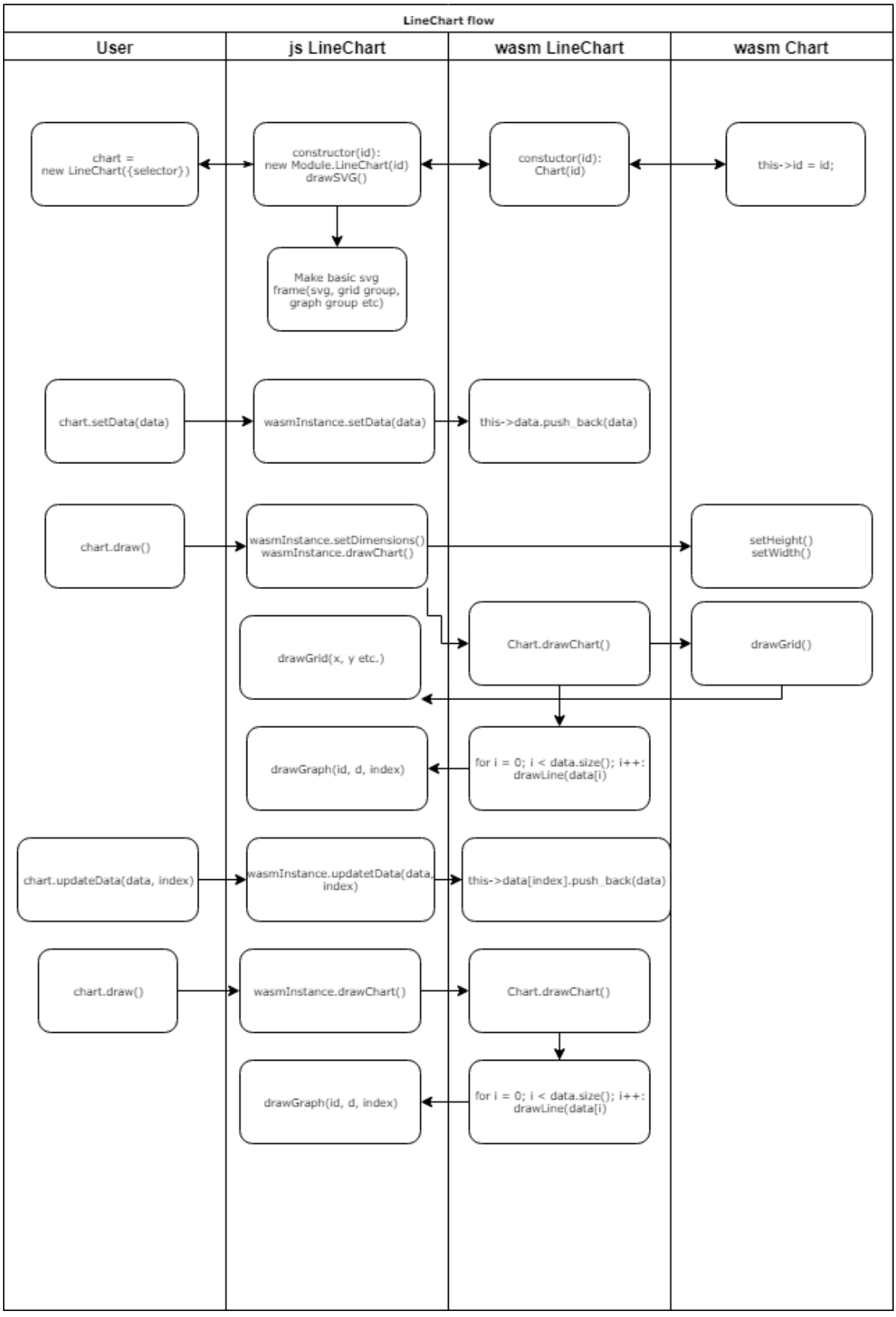
6.2.1 Ajaseeriatel põhineva joondiagrammi arhitektuur

Et luua uus joondiagramm loob kasutaja uue JavaScript „LineChart“ objekti, mis omakorda loob uue WebAssembly „LineCharti“ instantsi, ja salvestab selle väljale „wasmInstance“. Järgmiseks on vaja graafikule anda ette andmed, selleks kutsub kasutaja JavaScipti objekti funktsiooni „setData()“, mis töötleb andmed WebAssembly'le sobivale kujule (C++ vektor tüüp) ning edastab töödeldud andmed WebAssembly instantsile.

Järgmiseks peab kasutaja kutsuma JavaScripti objekti funktsiooni „draw()“. Kui võrreldes eelmise joonistamisega on graafiku mõõtmed muutunud, siis kõigepealt saadab JavaScipti object WebAssembly'le uued mõõtmed ning joonistatakse graafiku raamistik (tavaliselt esimesel joonistamisel). Edasi joonistatakse graafiku telgede väärtused ning sisestatud andmete põhjal joondiagrammid. WebAssembly instantsis käiakse andmed läbi ning genereeritakse SVG path elemendile d (sõne graafikupunktidest) atribuut ning kutsutakse JavaScripti funktsiooni „drawLineGraph()“, andes sellele kaasa graafiku identifikaatori, genereeritud d ning joone indeksi.

Diagrammi uuendamiseks tuleb kasutajal talitada sarnaselt loomisega. Erinevus seiseneb selles, et „setData()“ asemel tuleb kutsuda „updateData()“ funktsiooni, millele tuleb kaasa anda ka graafiku joone indeks (kui on üks joon, siis 0, kui rohkem, siis vastavalt andmete

sisestamise järjekorrale). Järgmiseks kutsub kasutaja „drawChart()“ funktsiooni ning diagramm joonistakse. Joondiagrammi loomise üldistatud voog on esitatud joonisel 7.



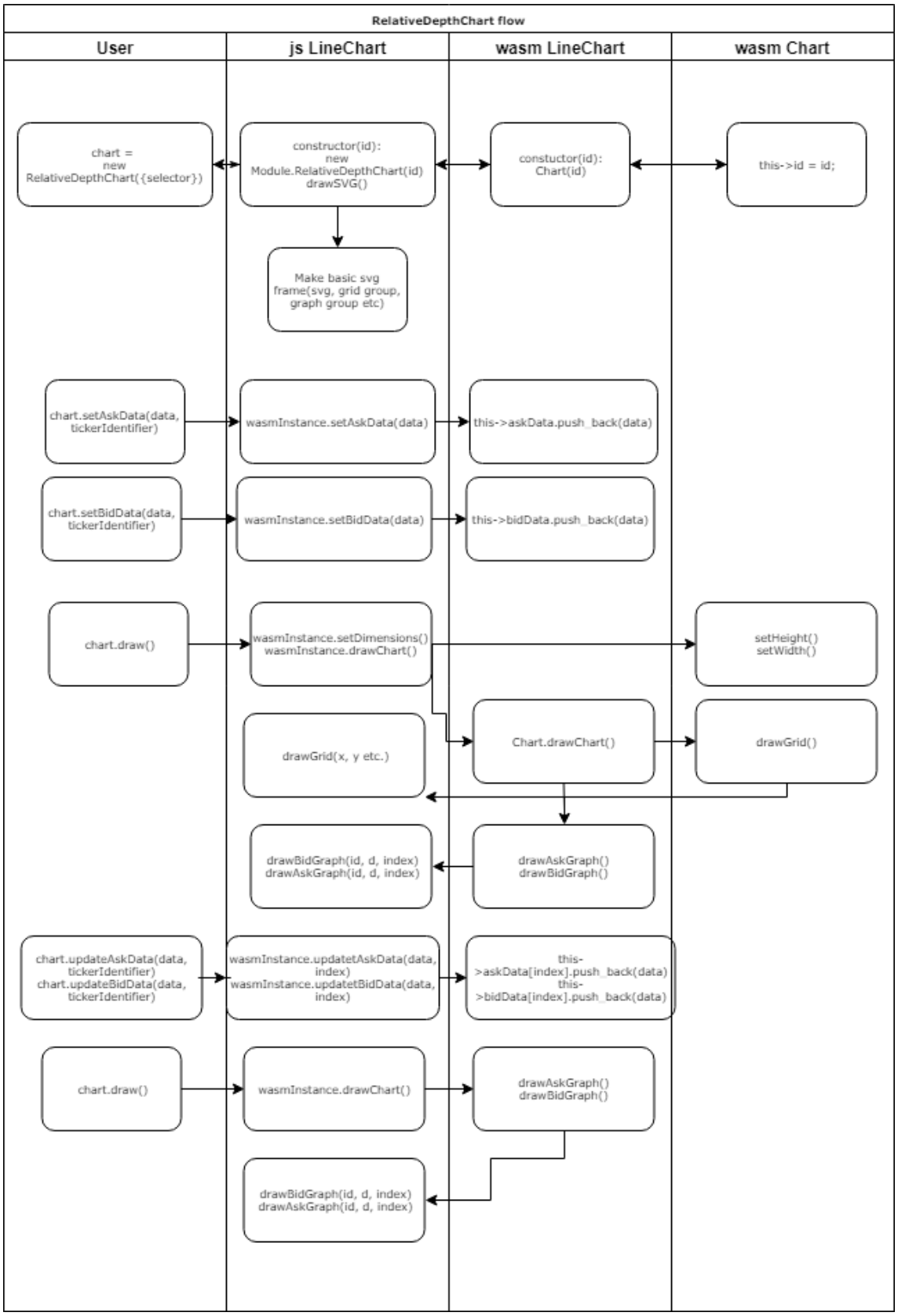
Joonis 7. Joondiagrammi loomise ja uuendamise üldistatud voog.

6.2.2 Tellimusraamatu suhtelise sügavuse graafiku arhitektuur

Et luua uus tellimusraamatu suhtelise sügavuse graafik loob kasutaja uue JavaScript „RelativeDepthChart“ objekti, mis omakorda loob uue WebAssembly „RelativeDepthChart“ instantsi, ja salvestab selle väljale „wasmInstance“. Andmete edastamiseks graafikule tuleb selle graafikutüübi puhul kutsuda kahte funktsiooni: „setAskData()“ ja „setBidData()“ (vastavalt müügi- ja ostutellimused), JavaScripti objektis töödeldakse taas andmed WebAssembly'le arusaavale kujule ning edastatakse need (C++ vektor tüüp).

Järgmiseks peab kasutaja kutsuma JavaScripti objekti funktsiooni „draw()“. Kui võrreldes eelmise joonistamisega on graafiku mõõtmed muutunud, siis kõigepealt saadab JavaScipti object WebAssembly'le uued mõõtmed ning joonistatakse graafiku raamistik (tavaliselt esimesel joonistamisel). WebAssembly käib andmed läbi ning genereerib SVG path elemendile d (sõne graafikupunktidest) atribuudi ning kutsub JavaScripti funktsioone „makeGraphLine()“ ja „makeGraphFill()“, andes sellele kaasa graafiku identifikaatori, genereeritud d ning tõeväärtuse isBid (tõene kui tegemist on ostutellimuse graafikuga). Selle tulemusena joonistatakse graafiku telgede väärtused ning ostu- ja müügitellimuste diagrammid.

Graafiku uuendamiseks tuleb kasutajal talitada sarnaselt loomisega. Erinevus seiseneb selles, et „setAskData()“ ja „setBidData()“ asemel kutsutakse „updateAskData()“ ja „updateBidData()“ funktsioone, millele tuleb kaasa anda ka kauplemispaari identifikaator (näiteks BTCUSD). Järgmiseks tuleb kutsuda „drawChart()“ funktsiooni ning diagramm joonistakse. Tellimusraamatu suhtelise sügavuse graafiku loomise üldistatud voog on esitatud joonisel 8.



Joonis 8. Tellimusraamatu suhtelise sügavuse loomise ja uuendamise üldistatud voog.

6.3 Graafikute teegi realisatsioon

Selles jaotises selgitan implementatsiooni põhilisi osasid ning toon välja programmikoodi ning kompilaatorikäsud, mis on vajalikud WebAssembly ja JavaScripti vahelise suhtluse võimaldamiseks.

6.3.1 WebAssembly klasside ja klassifunktsioonide JavaScriptis kasutatavaks tegemine

Selleks, et programmeerimis keelest C++ WebAssembly'sse kompileeritud andmetüübid, klassid ning klassifunkstioonid JavaScriptile kättesaadavaks teha tuleb kirjutada lisakoodi ning kompilaatorile lisada „–bind" lipp.

Esiteks tuleb kaasata Emscripteni API C++ koodis, kasutades koodirida:

```
#include <emscripten.h>
```

Joonis 9. Emscripteni API kaasamine C++ koodis.

Järgmiseks tuleb kaasata „EMSCRIPTEN_BINDINGS" koodiplokk, kuhu tuleb ette anda kõik klassid, nende konstruktorid ja funktsioonid, mida on tarvis JavaScriptis kasutada. Et eksportida klassi, klassikonstruktorit ja klassifunktsioone tuleb „EMSCRIPTEN_BINDINGS" ploki sisse defineerida klass, klassikonstruktor ja kõik eksporditavad klassifunkstioonid. Lisaks võimaldab „EMSCRIPTEN_BINDINGS" plokk teha JavaScriptis kasutatavaks C++ andmetüübi „std::vector<>" (võimaldab ka andmetüüpi „std::map<>", kuid seda antud töö loomisel ei kasutatud), mida on kasulik kasutada andmete saatmiseks WebAssembly moodulisse. Näidisdefinitsioonid koos selgituste ning JavaScriptis kasutamise näidetega on esitatud tabelis 8 ning antud töös kasutatud plokk on toodud joonisel 10.

Tabel 8. EMSCRIPTEN_BINDINGS ploki näidisdefinitsioonid.

Definitsioon	**Selgitus**	**Kasutamine JavaScriptis**
class_<TestClass>("Exported TestClass")	Definitsoon ekspordib C++ klassi „TestClass“ nimega „ExportedTestClass“.	const instance = new Module.ExportedTestClass().
.constructor<int>()	Lisab eksporditavale klassile konstruktori, mis võtab argumendina ühe täisarvulise muutuja.	const instance = new Module.ExportedTestClass(1337).
.function("testFunction", &TestClass:: testFunction)	Lisab eksporditavale klassile funktsiooni nimega „testFunction“.	instance.testFunction().
register_vector<int>("TestVector");	Ekspordib andmetüübi „std::vector<int>“ nimega „TestVector“.	const vector = new Module.TestVector(). Lisaks saab kasutada andmetüübi funktsioone, näiteks vector.push_back(1337), mis lisab andmetüübi lõppu uue täisarvu „1337“.

```
EMSCRIPTEN_BINDINGS(order_book_chart_class) {
        class_<Chart>("Chart")
                .function("getHeight", &Chart::getHeight)
                .function("getWidth", &Chart::getWidth)
                .function("setDimensions", &Chart::setDimensions)
                .function("drawCursor", &Chart::drawCursor)
        ;
        register_vector<std::vector<double>>("OrderBookDataVector");
        register_vector<double>("OrderBookDataPointVector");
        class_<LineChart, base<Chart>>("LineChart")
            .constructor<int>()
            .function("updateData", &LineChart::updateData)
            .function("setData", &LineChart::setData)
            .function("drawChart", &LineChart::drawChart)
            .function("drawTooltip", &LineChart::drawTooltip)
        ;
        class_<RelativeDepthChart, base<Chart>>("RelativeDepthChart")
            .constructor<int>()
            .function("updateOrderBookBidData",
&RelativeDepthChart::updateOrderBookBidData)
            .function("updateOrderBookAskData",
&RelativeDepthChart::updateOrderBookAskData)
            .function("setOrderBookAskData",
&RelativeDepthChart::setOrderBookAskData)
            .function("setOrderBookBidData",
&RelativeDepthChart::setOrderBookBidData)
            .function("drawChart", &RelativeDepthChart::drawChart)
            .function("drawTooltip", &RelativeDepthChart::drawTooltip)
        ;
}
```

Joonis 10. Implementatsioonis kasutatud EMSCRIPTEN_BINDINGS plokk.

6.3.2 JavaScript funktsioonide kutsumine WebAssembly moodulis

Et graafikuid joonistada oli töös vaja WebAssembly moodulis kutsuda JavaScripti funktsioone, et nende läbi HTML elemente muuta/lisada/eemaldada. Näiteks kui diagrammi koordinaadid on arvutatud, siis tuleb need JavaScripti vastavale funktsioonile edastada. Seda võimaldab Emscripten API funkstiooni „EM_ASM_“ näol, mis võimaldab kirjutada JavaScripti C++ koodis. „EM_ASM_“-ile saab ette anda primitiivseid muutujaid ning massiive nendest. Kuna SVG koordinaatide järjestus d on aga sõne tüüpi, siis tuleb see kõigepealt *char* massiiviks konverteerida ning seejärel, et seda JavaScriptis sõnena kasutada tuleb muutuja peal rakendada funktsiooni „UTF8ToString()“, mis konverteerib massiivi uuesti sõneks. Näide JavaScripti kasutamisest WebAssembly moodulis on toodud joonisel 11. Joonisel muutujad „$0“, „$1“ ja „$2“ on vastavas järjekorras ette antud väärtused.

```
EM_ASM_({
    drawLineGraph($0, UTF8ToString($1), $2);
}, this->id, d_char_array, index);
```

Joonis 11. JavaScripti kasutamine WebAssembly moodulis.

6.3.3 Visualiseeritavate andmete voog kasutajalt JavaScripti graafiku objektile

Et andmed visualiseerida peab kasutaja andmed kõige pealt saatma JavaScripti graafiku objektile, mis töötleb neid ja edastab WebAssembly moodulile. Mõlemad realiseeritud graafikutüübid võtavad vastu andmed objektide massiivina. Ajaseeriatel põhinev joondiagrammi andmepunkti vaikimisiformaat on „{date: 1555350433, value: 13.37}“, kus väli „date“ on ajatempel ning väli „value“ on väärtus sellel ajahetkel. Tellimusraamatu suhtelise sügavuse graafiku jaoks on andmete vaikimisiformaat: „{price: 13.37, size: 13.37}“, kus väli „price“ on kauplemispaari hind ning väli „size“ on tellimuse suurus selle hinna kohta.

Mõlema graafikutüübi puhul on võimalik andmete vaikimisiformaadid objekti loomisel üle kirjutada, andes sätete objektis ette vastavad väljad. Joondiagrammi puhul on väljadeks „dateKey“ ja „valueKey“ ning tellimusraamatu graafiku puhul „priceKey“, „volumeKey“. Graafiku objekti loomise näited sätetega on toodud joonisel 12.

```
Const orderBookChart = new RelativeDepthChart({selector: 'my-order-book-
chart', priceKey: 'myPrice', volumeKey: 'volume'});
Const lineChart = new LineChart({selector: 'my-line -chart', dateKey:
'timestamp', valueKey: 'price'});
```

Joonis 12. Graafiku objektide loomise näide, koos sätetega.

6.3.4 Visualiseeritavate andmete voog JavaScripti objektist WebAssembly moodulisse

Selleks, et kasutaja poolt JavaScriptile antud andmed visualiseerida tuleb need WebAssembly moodulile edasi saata. Andmete saatmise JavaScriptist WebAssembly moodulisse realiseerisin kasutades C++ andmetüüpi „std::vector<>“. Mõlema graafiku puhul loetakse andmepunktid vektorisse „std::vector<double, double>“ ning lisatakse need omakorda vektorisse, mis koosneb andmepunktide vektoritest. Kui andmed on vektorisse loetud, siis saadetakse vektor WebAssembly moodulisse ning kustutatakse vektor. Andmete saatmine WebAssembly moodulisse on toodud joonisel 13.

```
setData(data, index = -1) {
       const vec = new Module.DataVector();
        for (let point of data) {
             const pointVec = new Module.DataPointVector();
             pointVec.push_back(parseFloat(point[this.dateKey]) / 1000);
             pointVec.push_back(parseFloat(point[this.valueKey]));
             vec.push_back(pointVec);
        }
        this.wasmInstance.setData(vec, index);
        vec.delete();
}
```

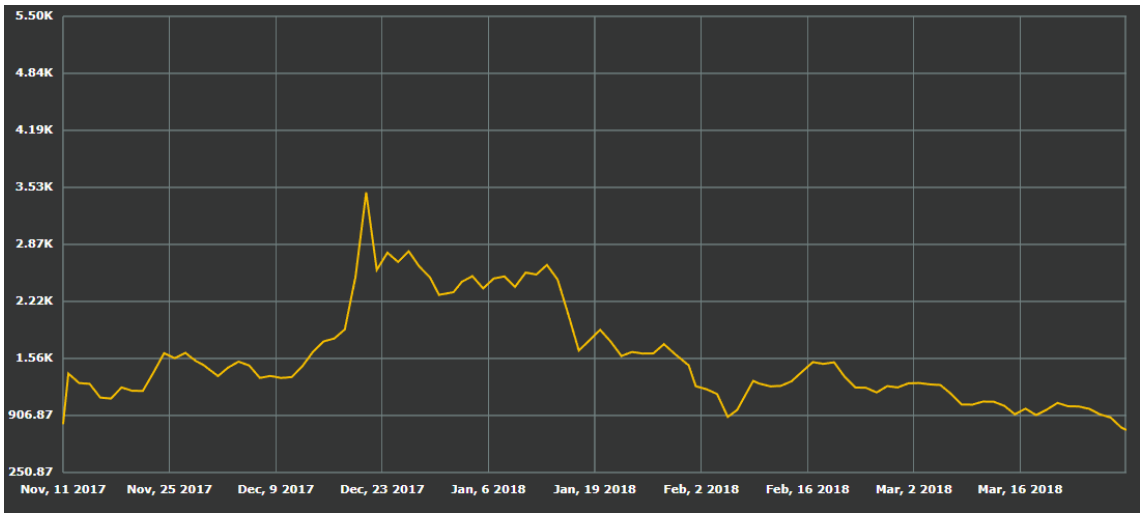
Joonis 13. Andmete saatmine JavaScripti objektist WebAssembly moodulisse

6.3.5 Ajaseeriatel põhinevate andmete joondiagrammil visualiseerimine

Kui andmed on jõudnud WebAssembly moodulisse, siis järgmine samm on nende visualiseerimine. Selleks tuleb käia läbi kõik andmepunktid ning nende põhjal arvutada SVG-le paigutatava joone koordinaadid. Loodud programmikood jagab andmed 100 võrdseks vahemikuks, võtab vahemikku jäävate punktide väärtuste aritmeetilised keskmised ning genereerib vastavalt 100 koordinaatpunktist koosneva sõne.

Kui sõne koordinaatpunktidest on genereeritud, siis kutsutakse JavaScripti funktsiooni „drawLineGraph()“, kuhu antakse ette graafiku unikaalne identifikaator, genereeritud sõne ning diagrammi indeks antud graafikul. JavaScripti funktsioon otsib vastavalt graafiku identifikaatorile ja diagrammi indeksile üles õige HTML elemendi ning seab selle d atribuudi väärtuseks genereeritud sõne. Kui vastavat elementi ei leitud, siis see luuakse.

Graafiku teljestik arvutatakse vastavalt graafiku laiusele ja kõrgusele ning teljestiku väärtused arvutatakse vastavalt andmepunktide minimaalsele ja maksimaalsele väärtusele. Näide ajaseeriatel põhinevast joondiagrammist on toodud joonisel 14. Joonisel on visualiseeritud krüptovaluuta Bitcoin Cash (BCH) hind Ameerika dollarites ajavahemikus 11.11.2018 – 31.03.2018.



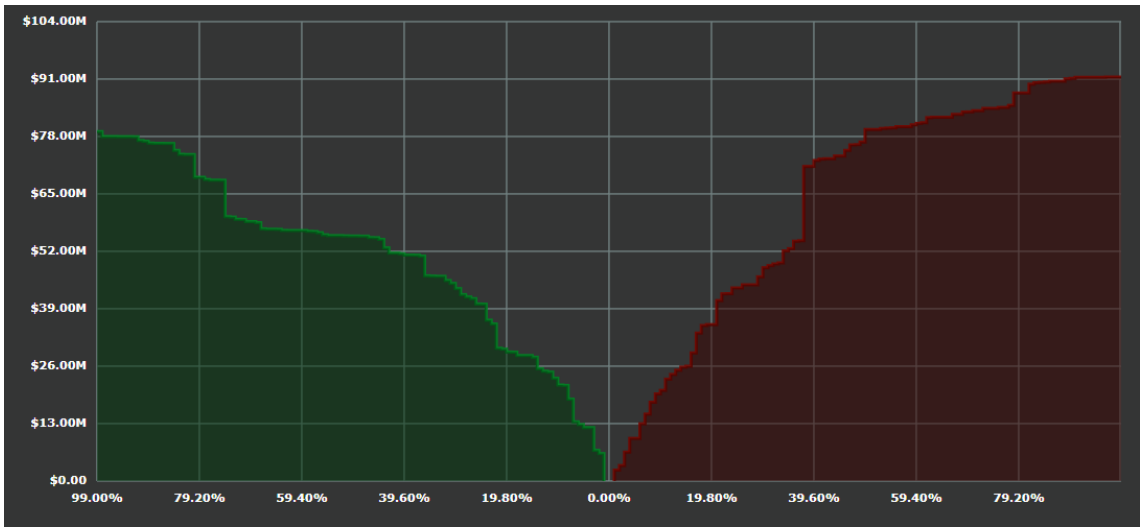
Joonis 14. Näide ajaseeriatel põhinevast joondiagrammist.

6.3.6 Tellimusraamatu andmete visualiseerimine suhtelise sügavuse graafikul

Kuna selle graafikutüübi eesmärk on visualiseerida mitme sama baasvaluutaga kauplemispaari tellimusraamatu sügavust, siis tuli leida ühine suurus, et seda graafikul kuvada. Selleks suuruseks valisin protsentuaalse erinevuse kauplemispaari hetkehinnast, kus hind arvutatakse aritmeetilise keskmisena kõige odavamast müügitellimuse hinnast ning kõige kallimast ostutellimuse hinnast. Kuna nõue on, et baasvaluuta peab olema sama, siis y-telje väärtuseks on tellimuste suuruste summa baasvaluutas.

Et andmed visualiseerida jagatakse nii müügi- kui ka ostutellimuste protsentuaalse vahemiku 100 osaks, näiteks kui maksimaalne hälve on 10%, siis on samm vastavalt 0.1%. Järgmiseks arvutatakse iga vahemiku jaoks tellimuste kogusuuruse baasvaluutas ning arvutatakse nende andmete põhjal koordinaatpunktidest koosneva sõne. Disaini huvides oli selle graafiku puhul vaja luua 4 SVG Path elementi, nii müügi- kui ka ostutellimuste jaoks üks joon ning teine sisu jaoks.

Teljestiku ja teljestikuväärtuste arvutamine käib samamoodi nagu joondiagrammi puhul. Näide tellimusraamatu suhtelise sügavuse graafikust on toodud joonisel 15. Joonisel on toodud vahenduskeskkonna hitBtc kõikide kauplemispaaride tellimusraamatute summa, mille baasvaluuta on Ameerika dollar (USD). Maksimaale hinna hälve hetke hinna suhtes on graafikul 99%. Andmed tulevad otse hitBtc veebisoklist.

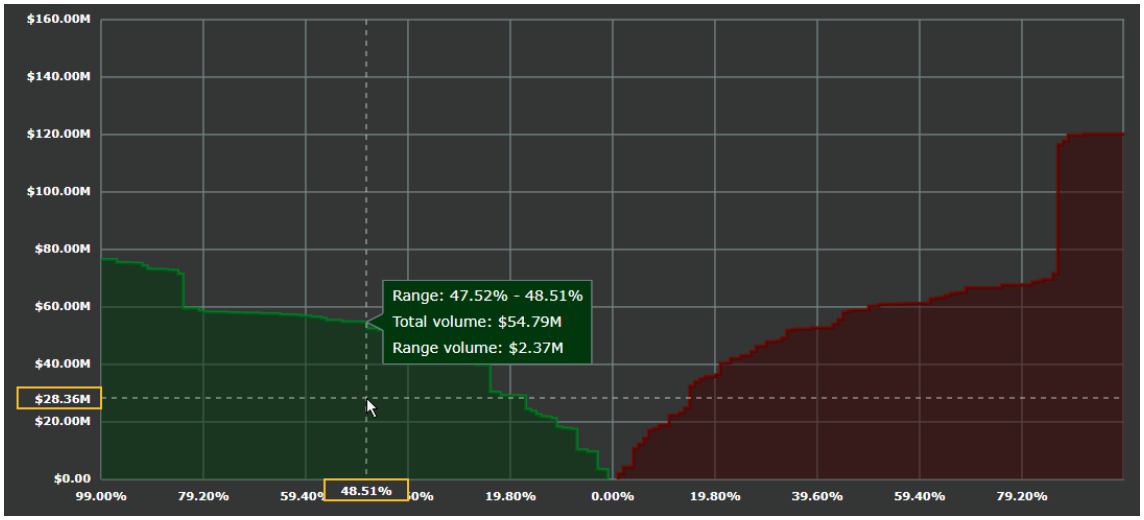


Joonis 15. Näide tellimusraamatu suhtelise sügavuse graafikust.

6.3.7 Graafikute lisafunktsionaalsus

Lisaks ülal kuvatule on kõik graafikud reaalajas uuendatavad. Kui andmed tulevad kuskilt veebisoklist on soovituslik luua JavaScriptis intervall, mis graafikut uuesti joonistab (näiteks iga 100 ms või 1 s järel), kuna andmeid tuleb peale nii kiirelt, et iga uuenduse korral graafikut joonistades on graafikut visuaalselt raske jälgida.

Lisaks on graafikutel realiseeritud kursori ja kursori punkti kohta andmete kuvamise funktsionaalsus. Näide kursorist ja andmete kuvamisest on toodud joonisel 16.



Joonis 16. Näide tellimusraamatu suhtelise sügavuse graafiku kursorist ja andmete kuvamisest.

7 Loodud teegi valideerimine nõutele

Loodud teegi nõuded ning tegelik tulemus on estitatud tabelis 9. Kõik nõuded (oodatav tulemus) on võetud töö peatükist 5.

Tabel 9. Loodud teegi oodatavad ja tegelikud tulemused.

Identifikaator	**Oodatav tulemus**	**Tegelik tulemus**
fn_1	Graafiku kogusuurus (kõrgus ja laius) peab olema võrdne selle konteineri suurusega.	Graafiku kogusuurus vastab konteineri suurusele. Graafiku kogusuuruse alla käivad lisaks graafiku osale endale ka teljestikust tingitud tühi ala.
fn_2	Graafikul peab olema visuaalselt vaadates arusaadav ning mõistlik joonestik, mis arvutatakse vastavalt graafiku suurusele (*grid*).	Kogu joonestik arvutatakse uuesti kui graafiku mõõtmed on muutunud ning graafiku joonestik on vastavalt minu kogemusele graafikute kasutamisega visuaalselt arusaadav ning mõistlik.
fn_3	Graafiku teljestikul peavad olema kuvatud tekstilised väärtused. Iga väärtus peab asuma vastava joonestiku joone keskel.	Teljetikul on kuvatud tekstilised väärtused ning need paiknevad vastava joonestiku joone keskel. Olenevalt graafiku tüübist on väärtuste tekstile lisatud ka ees- või järelliited (näiteks „%“).
fn_4	Graafik peab olema veebibrauseri akna suurusega kohanduv (*responsive*).	Graafik on kasutatav erineva suurusega brauseriakendes, kui pärast esmast loomist akna suurust ei muudeta. Hetkel puudub funktsionaalsus konteineri suuruse jälgmiseks.
fn_5	Graafiku arvutusosad peavad olema realiseeritud nii WebAssembly’s kui ka JavaScriptis, et saaks läbi viia võrdlusanalüüsi.	Kõik arvutusosad on realiseeritud nii WebAssembly’s kui ka JavaScriptis.
fn_6	Graafiku andmed peavad olema uuendatavad.	Mõlema graafikutüübi puhul on andmed uuendatavad kasutades vastavat uuendamise funktsiooni. Kui andmeid uuendada ja tahta graafikut nende andmetega kooskõlla

		viia tuleb kutsuda ka funktsiooni „draw()“
fn_7	Graafikul peab olema kursor, mis näitab kasutaja hiire kursori asukohta graafiku teljestike suhtes.	Graafikul on realiseeritud kahe katkendliku joonena kursor. Üks joon on vertikaalne ning teine horisontaalne. Mõlemad jooned algavad graafiku ühest servast ning lõppevad teises.
fn_8	Graafikul hiire kursoriga liikudes, peab kuvama info selles punktis olevate väärtuste kohta. Lisaks peab kuvama ka antud punktis teljestiku väärtust.	Graafikul hiirega liikudes kuvatakse antud punktis info. Küll vajab mõnes punktis info kuvamise väli kohandamist, sest see läheb graafikust välja (peamiselt nurkades). Teljestiku väärtused antud punkti kohta on kuvatud teljestikule ilmuvas väljas.
fn_9	Graafik peab toetama mitme joone lisamist.	Graafik toetab mitme joone lisamist. Mingit limiiti joonte arvu kohta seatud ei ole.
fn_10	Iga joone stiil (värv, paksus jne) peab olema valitav.	Hetkel see nõue täietud ei ole.
fn_11	Graafik peab toetama mitme kauplemispaari tellimusraamatu andmeid.	Graafikule saab lisada mitme kauplemispaari tellimusraamatu andmed. Mingeid piiranguid kauplemispaaride arvu kohta seatud ei ole.
fn_12	Ostutellimused peavad graafikul olema esitatud rohelisena ning müügitellimused punasena.	Ostutellimused on kuvatud graafikul vasakul pool rohelisena ning müügitellimused on kuvatud paremal pool punasena.
fn_13	Kui hiire kursor paikneb graafikul, siis graafikut uuesti ei tohi joonistata (Et kasutajal oleks võimalik konkreetsel ajahetkel järeldusi teha).	Kui hiire kursor paikneb graafikul, siis vaikimisi graafikut uuesti ei joonistata. Vaikimisi väärtuse saab üle kirjutada graafiku loomisel.
mfn_1	Kogu programmikood peab olema kommenteeritud ning vastama „puhta koodi“ põhimõtele.	Kogu programmikood on kommenteeritud ning on peetud kinni „puhta koodi“ põhimõtetest.
mfn_2	Graafikute loomise teek peab olema dokumenteeritud ning arendajatele arusaadavalt kasutatav.	Hetkel ei ole teek dokumenteeritud ning seega ka mitte arendajatel arusaadavalt kasutatav. Teeki ei avalikustata enne selle nõude täitmist.

Enamik teegile esitatud nõuetest said täidetud, kuid mitte kõik. Täitmata jäi dokumenteerituse nõue (mfn_2) ja graafiku elementide stiili valimise (CSS-i kasutamata) nõue (fn_10). Osaliselt jäi täitmata akna suurusega kohandumise nõue (fn_4) – puudub funktsionaalsus, mis tuvastab brauseriakna suuruse muutusi. Lisaks on osaliselt täidetud graafiku punktide kohta info kuvamise nõue (fn_8) – nurkades ei ole infoploki paigutus optimaalne. Ülejäänud nõuded on realisatsioonis täielikult täidetud

8 WebAssembly võrdlusanalüüs JavaScriptiga erineva suurusega andmestikkude korral

Vastavalt töö metoodika osas kirjeldatule teostasin mõõtmised mõlema graafikutüübi kohta.

8.1 Joondiagrammi võrdlusanalüüs

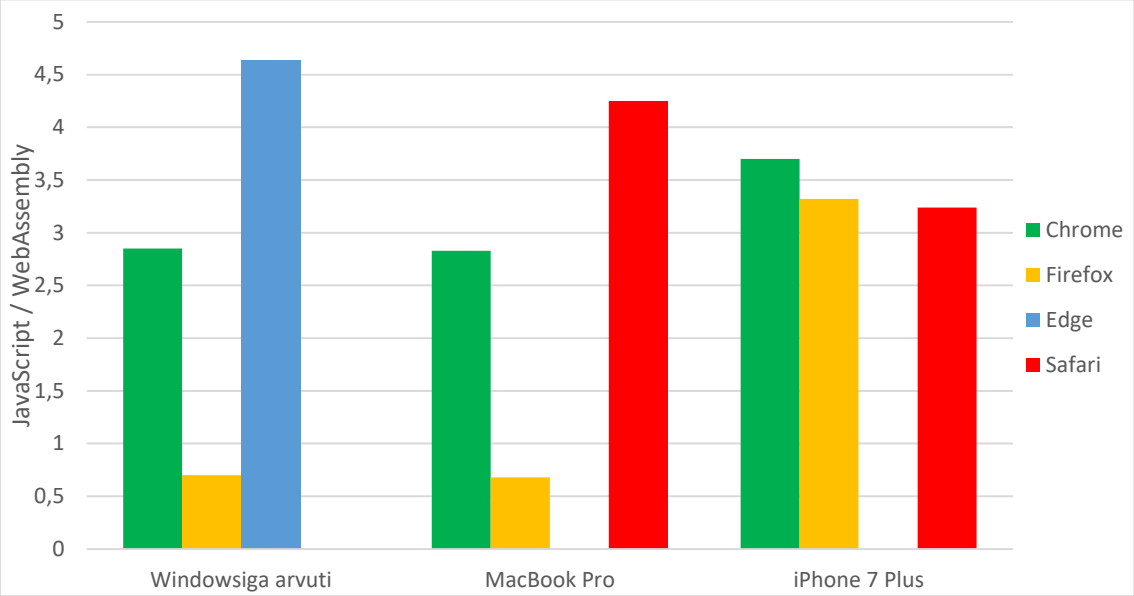
Joondiagrammi puhul mõõtsin kahte kõige rohkem aega nõudvat funktsionaalsust: andmete edastamine graafikule (funktsioon „setData()) ja graafiku joonistamine (funktsioon „draw()“). Kuna andmete edastamisel WebAssembly moodulile tuleb need ümber konverteerida, siis on loogiline hüpotees, et andmete edastamine graafikule toimub WebAssembly'ga lahenduse puhul aeglasemalt kui JavaScriptiga lahenduse puhul. Võrdlusanalüüsi tulemused on esitatud tabelis 10. Tabelis on kõikide mõõtmisiteratsioonide tulemused esitatud kõigil seadmetel ja veebibrauserites teostatud mõõtmiste aritmeetilise keskmisena.

Tabel 10. Joondiagrammi võrdlusanalüüsi tulemused.

Iteratsioon	**Andmepunktide arv**	**Andmete edastamine**	**Graafiku joonistamine**
line_1	100	0.45	1.05
line_2	1000	0.36	1.10
line_3	10000	0.35	2.23
line_4	100000	0.31	2.96
line_5	5000000	0.30	2.91

Vaadates saadud tulemusi tunduvad need oodatavad, kuid tegelikult esines mõningaid üllatusi. Nimelt mõlema arvuti peal (Windowsiga arvuti ja MacBook Pro) Firefoxis kahe viimase iteratsiooni (line_4 ja line_5) puhul JavaScripti graafiku joonistamine toimis kiiremini kui WebAssembly'ga. Selle olukorra põhjustas JavaScripti oluline kiiruse vahe teiste veebibrauseritega, näiteks Windowsiga arvuti puhul viimases iteratsioonis toimis Firefoxis JavaScript 3.77 korda kiiremini kui Chrome'is.

Kui McGilli ülikooli Sable'i uurimisrühma poolt läbi viidud uurimuses oli WebAssembly Firefoxis kõige kiirem, siis joondiagrammi joonistamisel mõlemas arvutis oli WebAssembly kiirus Firefoxis kõige suurem, jäädes alla nii Chrome'ile, Edge'ile kui ka Safarile. iPhone 7 Plusi peal oli Firefox siiski kiireim edestades Safarit 1.8% võrra ja Chrome'i 8.9% võrra. Graafiku joonistamise kiiruste suhted erinevate seadmete ja veebibrauserite korral on esitatud joonisel 17 (iteratsioon line_5). Kõik joondiagrammi võrdlusanalüüsi detailsed tulemused on esitatud lisas 2.



Joonis 17. JavaScriptiga lahenduse kiiruse suhe WebAssembly omasse.

Üldine tendents puhta JavaScriptiga lahenduse ning WebAssembly'ga lahenduse kiiruse suhtest oli, et andmepunktide arvu kasvuga suhe kasvas, kuid tasakaalustus viimase kahe iteratsiooni puhul. Oluline hüpe toimus teise ja kolmanda iteratsiooni vahel (1.10 vs 2.23). Mis puutub andmete edastamise kiirusesse, siis on näha, et andmepunktide arvu kasvuga muutub WebAssembly'ga lahendus JavaScripti omaga võrreldes järjest aeglasemaks. Praktikas tähendab see, et graafiku esmane kuvamine on WebAssembly'ga lahenduse korral aeglasem, kuid iga järgnev uuendus kiirem.

8.2 Tellimusraamatu suhtelise sügavuse graafiku võrdlusanalüüs

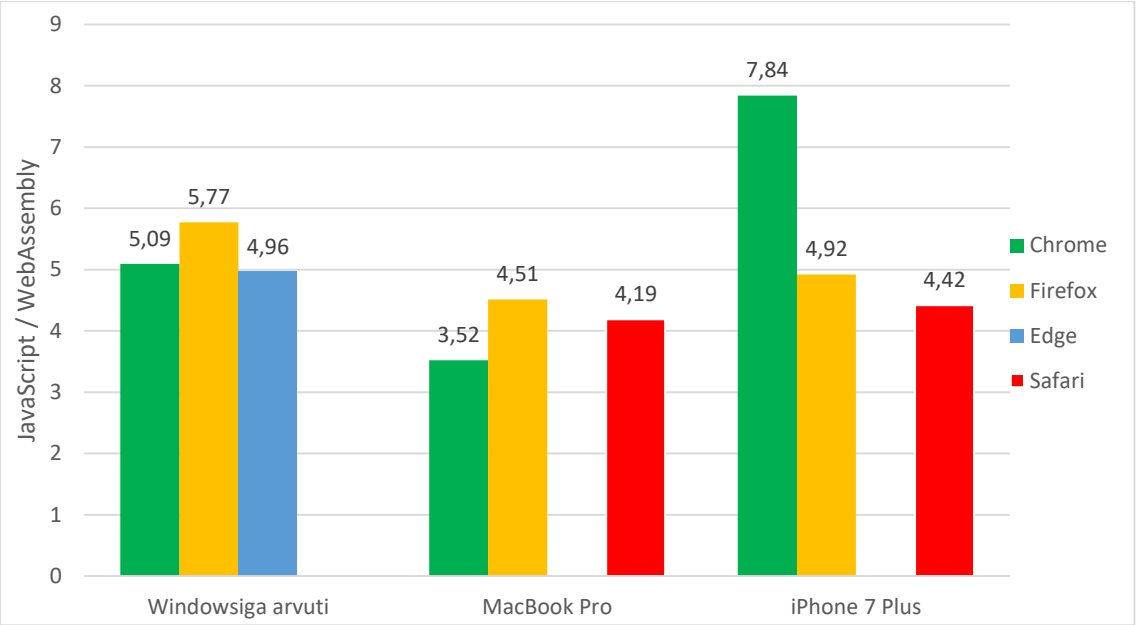
Tellimusraamatu suhtelise sügavuse graafiku puhul mõõtsin samuti kahte funktsionaalsust: andmete uuendamine (funktsioon „updateData()) ning graafiku joonistamine (funktsioon „draw()"). Andmete uuendamiseks konkreetse hinna kohta,

tuleb andmepunkt massiivist üles leida ning siis seda uuendada, mis tähendab, et andmestiku suuruse kasvuga kasvab ka operatsioonide arv, mis on vajalik uuenduse tegemiseks. Seega selle graafikutüübi jaoks on loogiline hüpotees, et mõlemad funktsioonid toimivad WebAssembly'ga lahenduse korral kiiremini. Kuna võrdlusanalüüsi iteratsioonide puhul on andmete uuendamine sama kulukas operatsioon (kogu tellimusraamatu andmed on alati olemas), siis seda mõõtmist iga iteratsiooni kohta eraldi ei teostanud. Võrdlusanalüüsi tulemused on esitatud tabelis 11. Tabelis on kõikide mõõtmisiteratsioonide tulemused esitatud kõigil seadmetel ja veebibrauserites teostatud mõõtmiste aritmeetilise keskmisena.

Tabel 11. Tellimusraamatu suhtelise sügavuse graafiku võrdlusanalüüsi tulemused.

Iteratsioon	**Hinna protsentuaalne hälve**	**Graafiku joonistamine**
depth_1	3%	2.04
depth_2	10%	2.45
depth_3	25%	3.57
depth_4	50%	3.94
depth_5	99%	5.02

Selle graafikutüübi võrdlusanalüüsi puhul üllatusi ei tekkinud, igas veebilehitsejas hinna protsentuaalse hälbe kasvatamisega JavaScripti ja WebAssembly kiiruse suhe kasvas. Andmete uuendamine toimis WebAssembly'ga lahenduse korral keskmiselt 18.01 korda kiiremini. Graafiku joonistamise kiiruste suhted erinevate seadmete ja veebibrauserite korral on esitatud joonisel 18 (iteratsioon depth_5). Võrdlusanalüüsi detailsed tulemused on toodud lisas 3.



Joonis 18. JavaScriptiga lahenduse kiiruse suhte WebAssembly omasse. (depth_5)

9 Järeldused

Selles peatükis esitan võrdlusanalüüsi põhjal järeldused ning pakun välja ning pakun välja sobilikud kasutusjuhud loodud teegile.

9.1 Järeldused joondiagrammi võrdlusanalüüsi põhjal

Selles jaotises teen joondiagrammi peal läbi viidud võrdlusanalüüsi tulemuste põhjal järeldusi.

9.1.1 Joondiagrammi joonistamine

Esimese kahe iteratsiooni (line_1 ja line_2) korral on graafiku joonistamise kiirus nii JavaSctiptiga lahenduse korral kui ka WebAssembly'ga lahenduse korral üsna võrdsed (suhted vastavalt 1.05 ja 1.10). Kuna andmestiku suuruse kasvuga siiski JavaScriptiga lahenduse kiiruse suhe WebAssembly'ga lahenduse kiirusesse kasvab (välja arvatud arvutite peal Firefoxi puhul), siis võib järeldada, et WebAssembly ja JavaScripti vaheline suhtlus ei tööta päris nii kiiresti kui JavaScripti osade omavaheline suhtlus.

Esimese kolme iteratsiooni puhul (line_1, line_2 ja line_3) toimib WebAssembly'ga lahendus küll kiiremini (line_3 puhul 2.23 korda), kuid sellise suurusega andmestikkude korral ei ole otstarbekas WebAssembly't kasutusele võtta. Nimelt ajakulu selle rakendamiseks ei kaalu üle sellega saavutatud kiiruse eelist. Kolmanda iteratsiooni (10000 andmepunkti) korral kulus JavaScriptiga lahendusel graafiku joonistamiseks keskmiselt kõige kauem 13.76 ms (iPhone 7 Plus Chrome). See on piisavalt väike aeg, et graafik oleks viivitusteta kasutatav.

Neljanda ja viienda iteratsiooni puhul tuleb WebAssembly kasutamine kindlasti kasuks (JavaScripti halvimad keskmised vastavalt 115 ms ja 479 ms iPhone'i peal, Chrome'is). Viienda iteratsiooni tulemus 479 ms ei pruugi tunduda küll pikk aeg, kuid kui graafikul on realiseeritud näiteks „kerimise" funktsionaalsus (ajavahemiku muutmine), siis igal kerimisel kuluks see aeg uuesti graafiku joonistamiseks. Sama loogika rakendub ka graafiku uuendamise puhul. Kasutaja seisukohast ei ole ligi 0.5 sekundi suurune viivitus

optimaalne, arvestades, et WebAssembly'ga lahenduse korral oleks vastav viivitus 129 ms.

Vastavalt mõõtmistulemustele võib väita, et andmestiku suuruse ja graafiku joonistamise kiiruse vahel on enam-vähem lineaarne seos (suurendades andmestikku 10 korda, kulub graafiku joonistamisele 10 korda rohkem aega). Seega kui andmestikku veelgi kasvatada suureneb vastavalt ka graafiku joonistamisele kuluv aeg (2 miljoni andmepunkti puhul JavaScriptiga lahenduse puhul eeldatav aeg ligi 2 sekundit ja WebAssembly'ga lahenduse puhul ligi 0.5 sekundit). Võib väita, et andmestiku suuruse kasvades WebAssembly kasulikkus suureneb (kiiruse vahe JavaScriptiga lahenduse ja WebAssembly'ga lahenduse vahel kasvab).

Mis puutub Firefoxi arvutite peal, siis valitud ülesande ja andmestikkude korral suudab Firefoxi JavaScripti mootor SpiderMonkey tunduvalt efektiivsemalt optimeerida vajalikud operatsioonid võrreldes teiste veebibrauserite JavaScripti mootoritega (Chrome'iga on vahe 3.77 kordne). Kuna selle teadustöö eesmärk ei ole erinevate veebibrauserite JavaScripti mootorite analüüs, siis antud teemat pikemalt ei lahka, kuid see teema väärib kindlasti uurimist mõnes teises teadustöös, sest vahe on märkimisväärne.

9.1.2 Joondiagrammi andmete saatmine

Vastupidiselt graafiku joonistamisele toimib andmete saatmine WebAssembly'ga lahenduse puhul aeglasemalt kui JavaScriptiga lahenduse puhul (erinevate suurustega andmestikkude korral on vahe keskmiselt 2.82 kordne). Olukord on tingitud sellest, et JavaScriptiga lahendusele saadetakse andmed samas formaadis, mis need algselt on, kuid WebAssembly'ga lahenduse korral töödeldakse andmed „std::vector<>" andmetüüpi.

Kiiruste vahe avaldub taas neljanda ja viienda iteratsiooni puhul (line_4, line_5), kus WebAssembly'ga lahenduse puhul on andmete saatmise suurim aeg vastavalt 409 ms ja 1834 ms (JavaScripti puhul vastavalt 145 ms ja 449 ms). Nagu graafiku joonistamise puhul on ka andmete saatmise kiirus enam-vähem lineaarselt seotud andmestiku suurusega, seega andmepunktide arvu kasvatamisel kahe kordseks suureneb ka andmete saatmise aeg 2 korda.

9.1.3 Soovitatavad kasutusjuhud

Kuna loodud rakenduse puhul andmete saatmise osa toimib WebAssembly'ga lahenduse puhul aeglasemalt, siis ei pruugi selle kasutamine igas olukorras otstarbekaks osutuda. Andmete saatmise aeg moodustab enamiku graafiku esmase kuvamise ajast („kerimisel" on andmed juba olemas, samuti uuendamisel on enamik andmeid tavaliselt olemas). Sellest aspektist lähtudes toon välja otstarbekad kasutusjuhud loodud teegi või mõne muu teegi kasutamiseks, kui andmepunkte on 100000 või rohkem.

Kõige otstarbekam on minu loodud teeki kasutada juhul, kui graafikut tuleb reaalajas pidevalt uuesti joonistada. Üks selline olukord on reaalajas uuendatav graafik, kuhu laaditakse andmeid järjest peale (näiteks otse veebisoklist). Teine näide on graafik, mida tuleb reaalajas pidevalt uuendada kasutaja poolt antud sisenditega („kerimine", suurendamine, mõne joone peitmine jne).

Kui graafik on mõeldud andmete staatiliseks kuvamiseks, see tähendab, et reaalajas seda ei uuendata ning kasutajal pole võimalik sellele sisendeid anda, siis eeldusel, et andmed on mõnele JavaScripti graafikute joonistamise teegile arusaadaval kujul, on soovituslik kasutada mõnda teist teeki. Sama kehtib ka olukorras, kus graafikut tuleb uuesti joonistada harva (näiteks minutilised intervallid), sest graafiku loomisele kulub vähem aega.

9.2 Järeldused tellimusraamatu suhtelise graafiku võrdlusanalüüsi põhjal

Selles jaotises teen tellimusraamatu suhtelise sügavuse graafiku peal läbi viidud võrdlusanalüüsi tulemuste põhjal järeldusi.

9.2.1 Tellimusraamatu suhtelise sügavuse graafiku joonistamine

Tellimusraamatu suhtelise sügavuse graafiku joonistamise võrdlusanalüüsi tulemused vastasid püsitatud hüpoteesile, ehk mida suurem on hinna protsentuaalne hälve (rohkem andmepunkte kaasatakse graafikule), seda suurem on kiiruse vahe WebAssembly'ga lahenduse ja JavaScriptiga lahenduse vahel. Kuna maksimaalne graafiku joonistamiseks kaasatud andmepunktide arv jäi vahemikku 50000 – 52000, siis ei tekkinud olukorda, millega JavaScript hakkama ei oleks saanud. Küll aga tuli kiiruse vahe kahe lahenduse puhul selgelt esile ning selle põhjal saab teha järeldusi olukordade kohta, kus

andmepunktide arv on suurem (näiteks andmed mitmed vahendusplatvormist, millest tuleb juttu töö edasi arenduste peatükis).

Analüüsides mõõtmistulemusi, siis minu hinnangul esimese 4 iteratsiooni (depth_1, depth_2, depth_3 ja depth_4) puhul WebAssembly kasutusele võtmine probleemi lahendamiseks märkimisväärset tulemust ei anna. JavaScripti lahendus suudab probleemi piisavalt efektiivselt lahendada nii, et kasutaja vaatenurgast kõik toimib ning viivitusi ei teki. Kõigis mainitud iteratsioonides jäi andmepunktide arv alla 40000.

Viienda iteratsiooni (depth_5, admepunktide arv vahemikus 50000 - 52000) juures võib WebAssembly kasulikuks ostutuda, olenevalt graafiku eesmärgist. Näieks kulus iPhone 7 Plusi peal Chrome'is graafiku joonistamisele JavaScriptiga lahenduse puhul 35 ms, mis tähendab seda et graafiku reaalajas sujuv uuendamine saab toimida 35 ms intervallidena (parimal juhul, arvestadeset tegemist on keskmisega, siis on intervall isegi suurem). Samas keskkonnas WebAssembly'ga lahendus puhul kujunes keskmiseks aga 4.48 ms (suhe 7.84 korda).

Võrdlusanalüüsist selgus, et JavaScripiga lahenduse puhul joonistamisele kuluv aeg on enam-vähem lineaarses sõltuvus kasutatavate andmepunktide arvuga. Seega kui andmepunktide arvu veelgi kasvatada (näiteks lisada juurde vahendusplatvorme – täpsemalt tuleb sellest juttu edasiarenduste osas), siis joonistamise aeg kasvab veelgi ning WebAssembly kasutamine probleemi lahendamiseks on mõistlik.

9.2.2 Tellimusraamatu suhtelise sügavuse graafiku andmete uuendamine

Selle graafiku tüübi puhul andmete uuendamine toimus WebAssembly'ga lahenduse korral küll 18.01 korda kiiremini, kuid praktikas minu arvates WebAssembly't selle pärast kasutusele võtta, sest andmete uuendamine võttis igas veebribrauseris aega alla 2 ms. Lisaks ei ole hetkel ka andmete uuendamise osa optimaalne, sest massiivi hakatakse algusest läbi käima, kuigi tegelikult kuna andmed on sorteeritud, siis optimaalne oleks kasutada binaarset otsingut. Küll aga saab järeldada, et massiivide läbi käimine on WebAssebmbly's tunduvalt efektiivsem kui JavaScriptis.

9.2.3 Soovitatavad kasutusjuhud

Loodud graafikutüüp on selles plaanis kasulik, et sarnast probleemi ükski teine teek hetkel ei lahenda. Kui kasutada mõnda muud teeki, siis peab arendaja ise tegelema kõigi valuutade andmete summeerimisega, mis nõuab lisa tööd.

WebAssembly kasutusele võtmine antud ülesande lahendamiseks on mõistlik, kui graafikut tuleb pidevalt reaalajas uuendada (väikse intervalliga, näiteks 50 ms). Kui on plaanis andmeid staatiliselt kuvada, ilma uuendamise fuktsionaalsuseta, siis JavaScript suudab probleem lahendada ning WebAssembly lisamisest tulenev lisa töö ning keerukus ei ole vajalik. Samuti ei ole vajalik WebAssembly't kasutusele võtta väikeste andmestikkude korral (alla 40000 andmepunkti). Üldiselt kehtib seos, et mida suurem on andmestik, seda rohkem WebAssembly end õigustab.

10 Võimalikud edasiarendused

Et loodud teek ka reaalselt veebirakenduste loomisel kasuks tuleks, siis tuleb teegile lisada funktsionaalsust, kirjutada dokumentatsioon ning olevasolevat funktsionaalsust optimeerida. Selles peatükis toon välja võimalikud edasiarendused (funktsionaalsed) teegile, mis on enne teegi avalikustamist vajalik ellu viia. Need aspektid selle töö raamesse ei mahtunud.

10.1 Joondiagrammi edasiarendused

Joondigrammi funktsionaalsuse kohapealt võib rääkida 3 suurimast hetkel realiseerimata osast: legend, erinevate joone sisse ja välja lülitamine ning graafiku „kerimine“. Lisaks tuleb anda teeki kasutavale arendajale võimalus mitmeid graafiku sätteid muuta (näiteks stiil, kursori olemasolu, graafiku uuendamine või mitte uuendamine, kui kursor asub graafikul jms).

Legend

Et graafikul olevad andmed kasutaja jaoks arusaadavad oleks, peab graafikul olema legend. Seda eriti juhul kui graafikul on mitu erinevat joont. Lisaks peab kasutajal olema võimalus joontele ise lihtalt vajalik stiil lisada. Stiili peab teek võimaldama lisada sisendina graafiku objektile, mitte ainult CSS-i kasutades. Kui legend on loodud, siis joonte sisse ja välja lülitamise funktsionaalsuse saab läbi selle lahendada. Vajutades legendil elemendile, lülitakse joon graafikul nähtavasse või peidetud režiimi.

Graafiku „kerimine“

Graafiku „kerimine“ hõlmab endast kahte funktsionaalsust, mis täidavad samat ülesannet: kerimisriba, kus saab graafikul kuvatavat ajavahemikku muuta ning graafiku suumimine, mis muudab graafikul kuvatavat ajavahemikku, kui kasutaja hiirt klikituna graafikul lohistab. Selle funktsionaalsuse lisamiseks tuleb ka olemasolevat funktsionaalsust veidi kohandada.

10.2 Tellmimusraamatu sügavuse graafiku edasiarendused

Tellimusraamatu graafiku puhul tuleb realiseerida mitme erineva andmeallika tugi, mis tähendab sisuliselt seda, et teeki kasutav arendaja saaks lisada andmeid mitmest erinevast veebisoklist lisatööd tegemata. Selleks peaks erinevatest andmeallikastest tulevad andmeid hoidma erinevates massiivides. Mõistlik oleks andmete massiivile lisada uus dimensioon.

Lisaks vajab parandamist olemasolev funktsionaalsus, näiteks graafikul kursoriga liikudes kuvatavad andmed konkreetse punkti kohta ei ole igas graafiku punktis optimaalselt paigutatud (nurkades võib minna graafikust välja). Mõistlik oleks lisada ka maksimaalse hinna dünaamilise muutmise tugi.

11 Kokkuvõte

Töö peamiseks eesmärgiks oli uurida, kas ja kui palju võimaldab WebAssembly kaasamine graafikute loomist veebirakendustes kiirendada erineva suurusega andmestikkude korral. Töö teiseks eesmärgiks oli kirjeldada töö käigus kujunenud töövoog ning selgitada, mida saaks teha paremini, kuna WebAssembly on uus tehnoloogia ning optimaalset töövoogu arendamiseks ei ole üheselt välja kujunenud.

Töö eesmärkide elluviimiseks loodi töö käigus graafikute joonistamise teek, milles on realiseeritud kahe graafikutüübi (joondigramm ja tellimusraamatu sügavuse graafik) põhiline funktsionaalsus. Kõik arvutusi nõudvad graafiku osad realiseeriti WebAssembly's ning sama funktsionaalsus dubleeriti ka JavaScriptis, et oleks võimalik läbi viia võrdlusanalüüs. Teegi osad, mis nõuavad HTML elementide manipuleerimist realiseeriti JavaScriptis. Võrdlusanalüüs viidi läbi mõlema graafikutüübi peal erineva suurusega andmestikkude korral.

Graafikute joonistamise teegi realiseerimise käigus tekkinud töövoog dokumenteeriti ning samuti kirjeldati arenduskeskkonna seadistamist ja tehnoloogiate valikut. Lisaks toodi välja valitud arenduskeskkonna puudused ning pakuti välja lahendus, mis võib osutuda mugavamaks. Töös kirjeldati tulemusena valminud graafikute teeki ning selle edasiarendusi, mis on vaja realiseerida enne teegi avalikustamist.

Enne tööd püsitatud hüpotees, et WebAssembly kasutamine võimaldab andmete visualiseerimist veebirakendustes kiirendada osutus tõeks. Võrdlusanalüüsist selgus, et joondiagrammide joonistamisel 500000 andmepunkti korral toimis graafiku joonistamine WebAssembly'ga lahenduse korral 2.91 korda kiiremini kui JavaScriptiga lahenduse korral. Tellimusraamatu suhtelise sügavuse graafiku joonistamine 50000 – 52000 andmepunkti korral toimis graafiku joonistamine WebAssembly'ga lahenduse korral 5.02 korda kiiremini kui JavaScriptiga lahenduse korral.

Graafikute loomise puhul on lisaks joonistamisele oluline ka andmete saatmine graafiku teegile ning WebAssembly moodulisse andmeid saates tuleb need ümber konverteerida. Joondidagrammi näitel selgus, et 500000 andmepunkti saatmine teegile toimis

JavaScriptiga lahenduse puhul 3.33 korda kiiremini kui WebAssembly'ga lahenduse korral. Seega graafiku esmane kuvamine toimib WebAssembly'ga lahenduse korral aeglasemalt kui JavaScriptiga lahenduse korral.

Võrdlusanalüüsi läbi viimisel arvestati, et graafiku joonistamisel on vajalikud nii WebAssembly arvutuste osa kui ka JavaScripti osa, mis tegelegeb HTML elementide manipuleerimisega. Seega võrdlusanalüüsi läbiviimisel jäeti sisse mõlemad osad, mitte ei võrreldud ainult arvutuste osa kiirust (seega oli sisse jäetud osade omavaheline suhtlus).

Kokkuvõtteks, töös valminud teek on algeline ning vajab edasiarendamist. Edasiarendamisega tegeletakse pärast töö ilmumist. Töö käigus selgus, et WebAssembly võimaldab graafikuid joonistada kiiremini kui puhas JavaScript. Loodud teek osutus kasulikuks kui graafikut on reaalajas tarvis pidevalt uuesti joonistada, kas läbi kasutaja poolt antud sisendite või andmete uuendamise. Kui andmeid on vaja kuvada staatiliselt, siis WebAssembly kasutamine ei osutunud mõistlikuks.

Kasutatud kirjandus

[1] „Can I Use,“ [Võrgumaterjal]. Available: https://caniuse.com/#search=webassembly. [Kasutatud 2 märts 2019].

[2] „WebAssembly ametlik koduleht,“ WebAssembly, [Võrgumaterjal]. Available: https://webassembly.org/. [Kasutatud 10 november 2018].

[3] M. Rouke, Learn WebAssembly. Build web applications with native performance using Wasm and C/C++, Packt Publishing Ltd, 2018.

[4] „NaCl and PNaCl,“ Google, [Võrgumaterjal]. Available: https://developer.chrome.com/native-client/nacl-and-pnacl. [Kasutatud 2 märts 2019].

[5] „asm.js,“ [Võrgumaterjal]. Available: http://asmjs.org/spec/latest/. [Kasutatud 4 aprill 2019].

[6] „WebAssembly High-Level Goals,“ WebAssembly, [Võrgumaterjal]. Available: https://webassembly.org/docs/high-level-goals/. [Kasutatud 10 november 2018].

[7] A. Zakai, „Why WebAssembly is Faster Than asm.js,“ 2017. [Võrgumaterjal]. Available: https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/. [Kasutatud 4 aprill 2019].

[8] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. Bastien ja M. Holman, „Bringing the Web up to Speed with WebAssembly,“ 2017. [Võrgumaterjal]. Available: https://people.mpi-sws.org/~rossberg/papers/Rossberg ,%20Titzer, %20Haas,%20Schuff,%20Gohman, %20Wagner,%20Zakai,%20Bastien,%20Holman%20-%20Bringing %20the%20Web%20up%20to %20Speed%20with %20WebAssembly%20[CACM].pdf. [Kasutatud 10 november 2018].

[9] „WebAssembly Semantics,“ WebAssembly, [Võrgumaterjal]. Available: https://webassembly.org/docs/semantics/. [Kasutatud 10 november 2018].

[10] A. Rossberg, „WebAssembly: high speed at low cost for everyone,“ 2016. [Võrgumaterjal]. Available: https://c10109cf-a-62cb3a1a -s-sites.googlegroups.com/ site/mlworkshoppe/2016 -1.pdf?attachauth =ANoY7cqo7VXO8h2fEI6sZDHIVYRR4TkSbYRf3h938R8-9d7SVeBmPMbNNb4AgFr93a9WdVLrN- 2i91dVwsYXiWoXYZGE4qy0l zNJQ8MHK0wKCcnKuDfGtVXrMFc3thdhry8Yk - iYHUvK5YPwVeQq6LCdDC. [Kasutatud 2 märts 2019].

[11] L. Clark, T. Schneidereit ja L. Wagner, „WebAssembly's post-MVP future: A cartoon skill tree,“ 2018. [Võrgumaterjal]. Available: https://hacks.mozilla.org/2018/10/webassemblys-post-mvp-future/. [Kasutatud 2 märts 2019].

[12] M. Bebenita, „WebAssembly is “30X” Faster than JavaScript,“ 2017. [Võrgumaterjal]. Available: https://medium.com/@mbebenita/webassembly-is - 30x-faster-than-javascript-c71ea54d2f96. [Kasutatud 10 november 2018].

[13] A. Turner, „WebAssembly Is Fast: A Real-World Benchmark of WebAssembly vs. ES6,“ 2018. [Võrgumaterjal]. Available: https://medium.com/@torch2424/webassembly-is-fast- a-real-world-benchmark-of-webassembly-vs-es6-d85a23f8e193. [Kasutatud 2 märts 2019].

[14] D. Herrera, H. Chen, E. Lavoie ja L. Hendren, „WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices,“ 2018. [Võrgumaterjal]. Available: http://www.sable.mcgill.ca/publications/techreports/2018-2/techrep.pdf. [Kasutatud 10 november 2018].

[15] Sable Lab, „Ostrich Benchmark Suite,“ [Võrgumaterjal]. Available: https://github.com/Sable/Ostrich. [Kasutatud 10 november 2018].

[16] A. Jangda, A. Guha, B. Powers ja E. Berger, „Mind the Gap: Analyzing the Performance of WebAssembly vs. Native Code,“ 2019. [Võrgumaterjal]. Available: https://arxiv.org/pdf/1901.09056.pdf. [Kasutatud 2 märts 2019].

[17] „BROWSIXi ametlik koduleht,“ BROWSIX, [Võrgumaterjal]. Available: https://browsix.org/. [Kasutatud 2 märts 2019].

[18] B. Malle, N. Giuliani, P. Kieseberg ja A. Holzinger, „The Need for Speed of AI Applications Performance Comparison of Native vs. Browser-based Algorithm Implementations,“ 2018. [Võrgumaterjal]. Available: https://arxiv.org/pdf/1802.03707.pdf. [Kasutatud 2 märts 2019].

[19] K. Ball, „How WebAssembly is Accelerating the Future of Web Development,“ 2018. [Võrgumaterjal]. Available: https://zendev.com/2018/06/26/webassembly - accelerating-future-web-development.html. [Kasutatud 2 märts 2019].

[20] S. Klabnik, „Is WebAssembly the return of Java Applets \& Flash?,“ 2018. [Võrgumaterjal]. Available: https://words.steveklabnik.com/is-webassembly -the-return-of-java-applets-flash. [Kasutatud 2 märts 2019].

[21] „WebAssembly roadmap,“ WebAssembly, [Võrgumaterjal]. Available: https://webassembly.org/roadmap. [Kasutatud 10 november 2018].

[22] „WebAssembly Minimum Viable Product,“ WebAssembly, [Võrgumaterjal]. Available: https://webassembly.org/docs/mvp/. [Kasutatud 10 november 2018].

[23] „Adobe Photoshop system requirements,“ [Võrgumaterjal]. Available: https://helpx.adobe.com/photoshop/system-requirements.html. [Kasutatud 4 aprill 2019].

[24] B. Smith, „WebAssembly Garbage collection,“ 2018. [Võrgumaterjal]. Available: https://github.com/WebAssembly/proposals/issues/16. [Kasutatud 2 märts 2019].

[25] „WebAssembly studio,“ [Võrgumaterjal]. Available: https://webassembly.studio/. [Kasutatud 2 märts 2019].

[26] „Emscripteni ametlik kodulehekülg,“ Emscripten, [Võrgumaterjal]. Available: https://emscripten.org/. [Kasutatud 10 november 2018].

[27] „Order Book,“ [Võrgumaterjal]. Available: https://www.investopedia.com/terms/o/order-book.asp. [Kasutatud 4 aprill 2019].

[28] „About HTML5 WebSocket,“ [Võrgumaterjal]. Available: https://www.websocket.org/aboutwebsocket.html. [Kasutatud 4 aprill 2019].

[29] „AmCarts live order book / depth chart,“ [Võrgumaterjal]. Available: https://www.amcharts.com/demos/live-order-book-depth-chart/. [Kasutatud 2 märts 2019].

[30] „D3.js,“ [Võrgumaterjal]. Available: https://d3js.org/. [Kasutatud 2 märts 2019].

[31] „D3 Market Depth Chart built from Order Book,“ 2018. [Võrgumaterjal]. Available: https://bl.ocks.org/idibidiart/42e8abf6fde52f54cec58064f9fd5582. [Kasutatud 2 märts 2019].

[32] „HitBTC,“ [Võrgumaterjal]. Available: https://hitbtc.com/BTC-to-USDT. [Kasutatud 4 aprill 2019].

[33] „Compiling a New C/C++ Module to WebAssembly,“ [Võrgumaterjal]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly/C_to_wasm. [Kasutatud 2 märts 2019].

Lisa 1 – Näide WASM-ist

```
0x00000000 0061736D01000000010A026000006002 .asm.......`..`.
0x00000010 7F7F017F03030200010405017001010 ............p...
0x00000020 05030100020615037F01418088040B7F ..........A.....
0x00000030 00418088040B7F004180080B07360406 .A......A....6..
0x00000040 6D656D6F727902000B5F5F686561705F memory..__heap_
0x00000050 6261736503010A5F5F646174615F656E base..__data_en
0x00000060 6403020E61646454776F496E74656765 d...addTwoIntege
0x00000070 727300010A0C0202000B070020012000 rs.......... . .
0x00000080 6A0B00640B2E64656275675F696E666F j..d..debug_info
0x00000090 5400000004000000000040100000000 T..............
0x000000A0 0C002300000000000000430000000500 ..#.......C.....
0x000000B0 000007000000020500000070000005C ..............\
0x000000C0 0000000104500000000036F0000000104 .....P....o.....
0x000000D0 500000000371000000010450000000000 P....q.....P....
0x000000E0 046B000000050400001 00E2E64656275 .k..........debu
0x000000F0 675F6D6163696E666F00004F0D2E6465 g_macinfo..O..de
0x00000100 6275675F6162627265760111012 50E13 bug_abbrev...%..
0x00000110 05030E10171B0E110112060000022E01 ..............
0x00000120 11011206030E3A0B3B0B271949133F19 ......:.;.'.I.?.
0x00000130 0000030500030E3A0B3B0B4913000004 .......:.;.I....
0x00000140 2400030E3E0B0B0B00000000620B2E64 $...>.......b..d
0x00000150 656275675F6C696E6552000000040037 ebug_lineR.....7
0x00000160 000000010101FB0E0D00010101010000 ..............
0x00000170 00010000012F746D702F6275696C645F ...../tmp/build_
0x00000180 356B7638643132637530632E24000066 5kv8d12cu0c.$..f
0x00000190 696C652E6300010000000000502050000 ile.c..........
0x000001A0 0015050B0A2105020658020100010100 .....!...X......
0x000001B0 7E0A2E64656275675F737472636C616E ~..debug_strclan
0x000001C0 672076657273696F6E20382E302E3020 g version 8.0.0
0x000001D0 287472756E6B20333431393630 29002F (trunk 341960)./
0x000001E0 746D702F6275696C645F356B76386431 tmp/build_5kv8d1
0x000001F0 32637530632E242F66696C652E63002F 2cu0c.$/file.c./
0x00000200 746D702F6275696C645F356B76386431 tmp/build_5kv8d1
0x00000210 32637530632E240061646454776F496E 2cu0c.$.addTwoIn
0x00000220 74656765727300696E74006100620000 tegers.int.a.b..
0x00000230 2B046E616D6501240200115F5F776173 +.name.$..__was
0x00000240 6D5F63616C6C5F63746F7273010E6164 m_call_ctors..ad
0x00000250 6454776F496E746567657273        dTwoIntegers
```

Lisa 2 – Joondiagrammi võrdlusanalüüsi detailsed tulemused

Iteratsioon	Seade	Brauser	setData()	Draw()
line_1	Windowsiga arvuti	Chrome	0.39 (0.33 vs 0.13)	1.32 (1.93 vs 2.54)
line_2	Windowsiga arvuti	Chrome	0.38 (2.60 vs 0.98)	1.29 (2.10 vs 2.70)
line_3	Windowsiga arvuti	Chrome	0.39 (24.75 vs 9.69)	1.79 (3.10 vs 5.54)
line_4	Windowsiga arvuti	Chrome	0.36 (255.89 vs 92.00)	3.09 (18.03 vs 55.15)
line_5	Windowsiga arvuti	Chrome	0.37 (1266.24 vs 464.13)	2.85 (91.39 vs 260.87)
line_1	Windowsiga arvuti	Firefox	0.54 (0.24 vs 0.13)	1.23 (1.50 vs 1.85)
line_2	Windowsiga arvuti	Firefox	0.45 (2.05 vs 0.92)	1.09 (1.61 vs 1.75)
line_3	Windowsiga arvuti	Firefox	0.41 (21.53 vs 8.75)	1.10 (2.94 vs 3.24)
line_4	Windowsiga arvuti	Firefox	0.35 (215.90 vs 18.90)	0.87 (18.89 vs 16.51)
line_5	Windowsiga arvuti	Firefox	0.35 (1156.50 vs 404.89)	0.70 (98.61 vs 69.24)
line_1	Windowsiga arvuti	Edge	0.31 (0.53 vs 0.17)	1.20 (2.66 vs 3.20)
line_2	Windowsiga arvuti	Edge	0.32 (3.48 vs 1.12)	1.31 (2.42 vs 3.16)
line_3	Windowsiga arvuti	Edge	0.35 (30.62 vs 10.68)	2.56 (3.47 vs 9.14)
line_4	Windowsiga arvuti	Edge	0.34 (301.37 vs 102.43)	4.63 (19.55 vs 90.46)
line_5	Windowsiga arvuti	Edge	0.34 (1517.46 vs 516.65)	4.64 (95.67 vs 444.02)
line_1	MacBook Pro	Chrome	0.63 (0.30 vs 0.19)	1.29 (1.63 vs 2.10)
line_2	MacBook Pro	Chrome	0.36 (2.32 vs 0.84)	1.16 (1.99 vs 2.30)
line_3	MacBook Pro	Chrome	0.38 (22.42 vs 8.42)	1.72 (2.69 vs 4.61)
line_4	MacBook Pro	Chrome	0.35 (227.74 vs 80.81)	2.77 (17.73 vs 49.06)
line_5	MacBook Pro	Chrome	0.34 (1190.59 vs 403.83)	2.83 (81.64 vs 231.03)
line_1	MacBook Pro	Firefox	0.53 (0.21 vs 0.11)	1.07 (1.24 vs 1.32)
line_2	MacBook Pro	Firefox	0.44 (1.8 vs 0.80)	1.05 (1.34 vs 1.40)
line_3	MacBook Pro	Firefox	0.39 (19.37 vs 7.57)	1.05 (2.36 vs 2.48)
line_4	MacBook Pro	Firefox	0.37 (203.36 vs 75.50)	0.76 (17.06 vs 13.00)
line_5	MacBook Pro	Firefox	0.40 (982.05 vs 396.65)	0.68 (85.55 vs 58.45)
line_1	MacBook Pro	Safari	0.56 (0.28 vs 0.16)	0.89 (1.22 vs 1.08)
line_2	MacBook Pro	Safari	0.34 (2.30 vs 0.78)	1.37 (1.24 vs 1.70)
line_3	MacBook Pro	Safari	0.28 (23.98 vs 6.74)	3.55 (2.25 vs 7.97)
line_4	MacBook Pro	Safari	0.18 (347.86 vs 63.51)	4.55 (15.11 vs 68.68)
line_5	MacBook Pro	Safari	0.18 (1768.80 vs 319.10)	4.25 (76.05 vs 323.50)
line_1	iPhone 7 Plus	Chrome	0.54 (0.33 vs 0.18)	0.77 (1.51 vs 1.17)
line_2	iPhone 7 Plus	Chrome	0.32 (2.83 vs 0.92)	1.18 (1.67 vs 1.97)
line_3	iPhone 7 Plus	Chrome	0.27 (43.37 vs 11.85)	3.13 (4.39 vs 13.76)
line_4	iPhone 7 Plus	Chrome	0.35 (408.96 vs 144.56)	3.57 (32.16 vs 114.82)
line_5	iPhone 7 Plus	Chrome	0.24 (1834.00 vs 448.70)	3.70 (129.30 vs 479.00)
line_1	iPhone 7 Plus	Firefox	0.29 (0.35 vs 0.10)	0.82 (1.45 vs 1.19)
line_2	iPhone 7 Plus	Firefox	0.33 (2.80 vs 0.92)	1.23 (1.65 vs 2.02)

line_3	iPhone 7 Plus	Firefox	0.34 (30.29 vs 10.20)	2.70 (3.66 vs 9.89)
line_4	iPhone 7 Plus	Firefox	0.21 (417.64 vs 88.64)	3.27 (29.08 vs 95.10)
line_5	iPhone 7 Plus	Firefox	0.24 (1523.00 vs 372.40)	3.32 (118.10 vs 392.50)
line_1	iPhone 7 Plus	Safari	0.34 (0.33 vs 0.11)	0.88 (1.37 vs 1.20)
line_2	iPhone 7 Plus	Safari	0.32 (2.80 vs 0.89)	1.20 (1.56 vs 1.88)
line_3	iPhone 7 Plus	Safari	0.33 (30.17 vs 10.09)	2.50 (3.69 vs 9.21)
line_4	iPhone 7 Plus	Safari	0.23 (381.60 vs 89.60)	3.17 (31.56 vs 99.94)
line_5	iPhone 7 Plus	Safari	0.24 (1466.70 vs 356.20)	3.24 (118.70 vs 384.30)

Lisa 3 – Tellimusraamatu graafiku võrdlusanalüüsi detailsed tulemused

Iteratsioon	Seade	Brauser	updateData()	Draw()
depth_1	Windowsiga arvuti	Chrome	10.27 (0.07 vs 0.71)	1.66 (2.55 vs 4.23)
depth_2	Windowsiga arvuti	Chrome		2.23 (2.69 vs 6.01)
depth_3	Windowsiga arvuti	Chrome		3.18 (2.76 vs 8.78)
depth_4	Windowsiga arvuti	Chrome		3.71 (3.18 vs 11.80)
depth_5	Windowsiga arvuti	Chrome		5.09 (3.07 vs 15.63)
depth_1	Windowsiga arvuti	Firefox	20.98 (0.06 vs 1.15)	2.25 (1.76 vs 3.97)
depth_2	Windowsiga arvuti	Firefox		2.45 (2.00 vs 4.91)
depth_3	Windowsiga arvuti	Firefox		3.69 (1.99 vs 7.33)
depth_4	Windowsiga arvuti	Firefox		4.08 (2.35 vs 9.58)
depth_5	Windowsiga arvuti	Firefox		5.77 (2.34 vs 13.5)
depth_1	Windowsiga arvuti	Edge	7.57 (0.08 vs 0.61)	1.97 (2.47 vs 4.85)
depth_2	Windowsiga arvuti	Edge		2.23 (2.62 vs 5.82)
depth_3	Windowsiga arvuti	Edge		3.33 (2.65 vs 8.81)
depth_4	Windowsiga arvuti	Edge		3.74 (3.09 vs 11.59)
depth_5	Windowsiga arvuti	Edge		4.96 (3.08 vs 15.27)
depth_1	MacBook Pro	Chrome	10.46 (0.06 vs 0.59)	1.4 (3.21 vs 4.51)
depth_2	MacBook Pro	Chrome		1.68 (3.54 vs 5.93)
depth_3	MacBook Pro	Chrome		2.31 (3.57 vs 8.24)
depth_4	MacBook Pro	Chrome		2.65 (3.99 vs 10.58)
depth_5	MacBook Pro	Chrome		3.52 (4.00 vs 14.10)
depth_1	MacBook Pro	Firefox	18.61 (0.06 vs 1.04)	1.74 (2.28 vs 3.97)
depth_2	MacBook Pro	Firefox		2.16 (2.58 vs 5.56)
depth_3	MacBook Pro	Firefox		3.20 (2.49 vs 7.94)
depth_4	MacBook Pro	Firefox		3.44 (3.018 vs 10.39)
depth_5	MacBook Pro	Firefox		4.51 (2.98 vs 13.42
depth_1	MacBook Pro	Safari	23.29 (0.03 vs 0.79)	1.85 (1.88 vs 3.48)
depth_2	MacBook Pro	Safari		2.15 (2.15 vs 4.63)
depth_3	MacBook Pro	Safari		3.01 (2.18 vs 6.57)
depth_4	MacBook Pro	Safari		3.34 (2.53 vs 8.44)
depth_5	MacBook Pro	Safari		4.19 (2.57 vs 10.76)
depth_1	iPhone 7 Plus	Chrome	30.49 (0.06 vs 1.94)	3.35 (2.49 vs 8.33)
depth_2	iPhone 7 Plus	Chrome		4.54 (2.65 vs 12.01)
depth_3	iPhone 7 Plus	Chrome		6.99 (2.82 vs 19.73)
depth_4	iPhone 7 Plus	Chrome		7.21 (3.39 vs 24.43)

depth_5	iPhone 7 Plus	Chrome		7.84 (4.48 vs 35.12)
depth_1	iPhone 7 Plus	Firefox	20.29 (0.07 vs 1.37)	2.10 (2.53 vs 5.32)
depth_2	iPhone 7 Plus	Firefox		2.18 (3.12 vs 6.81)
depth_3	iPhone 7 Plus	Firefox		3.27 (3.41 vs 11.16)
depth_4	iPhone 7 Plus	Firefox		3.80 (2.92 vs 11.10)
depth_5	iPhone 7 Plus	Firefox		4.92 (2.86 vs 14.10)
depth_1	iPhone 7 Plus	Safari	20.17 (0.07 vs 1.34)	2.08 (2.36 vs 4.90)
depth_2	iPhone 7 Plus	Safari		2.45 (2.42 vs 5.94)
depth_3	iPhone 7 Plus	Safari		3.16 (3.93 vs 12.42)
depth_4	iPhone 7 Plus	Safari		3.45 (4.85 vs 16.73)
depth_5	iPhone 7 Plus	Safari		4.42 (5.27 vs 23.28)

Lisa 4 – lähtekoodi hoidla Gitlabis

https://gitlab.cs.ttu.ee/Hendrik.Laas/magister.git