TARTU JAAN POSKA GÜMNAASIUM

AUTOR: GREGOR EESMAA

12.C KLASS

GALAKTIKATE FILAMENTIDE RUUMIJAOTUSE VISUALISEERIMINE

JUHENDAJAD: PhD ELMO TEMPEL, EINIKE REINVELT

SISSEJUHATUS

Galaktikate jaotus universumis ei ole ühtlane. Selles esineb nii tihedamaid kui ka hõredamaid kohti. Piltlikult moodustub galaktikatest ämblikuvõrgulaadne kärjestruktuur. Galaktikate visualiseerimine arvuti abil on üks paremaid meetodeid seda uurida. Seejuures osutub kõige tähtsamaks tarkvara oskuslik optimeerimine, vältides pikka ooteaega visualisatsiooniga opereerides. Käesoleva uurimistöö eesmärgiks on selline programm valmistada.

Teema valik tuleneb mitmest aspektist. Esiteks lihtsalt kasutatavat, vabavaralist ning optimeeritud galaktikate visualiseerimisprogrammi vabas vormis andmehulkade jaoks ei ole varasemalt loodud. Teiseks huvist kosmose (sh kosmoloogia ja astronoomia) vastu ning kolmandaks soov õppida kasutama ettevõtluses levinud programmeerimist hõlbustavat tarkvara.

Uurimistöö viiakse läbi, kasutades paindliku arendustsükli põhimõtteid. Tarkvara loomisel kasutatakse Processing teeki Javale, et lihtsustada graafikakaardi kasutust programmi töö kiirendamiseks. Teiste programmeerimiskeeltega võrreldes on Java kasutus õigustatud, kuna selle lähtekood muudetakse enne programmi jooksutamist masinale lihtsalt loetavaks koodiks, mistõttu on ta oluliselt kiirem näiteks Pythonist, mille lähtekoodi töödeldakse alles jooksutamise ajal. Samuti, võrreldes sarnase kiirusega C-keeltega on Javaga lihtsam prototüüpida, kuna ei pea muretsema näiteks mäluhalduse pärast.

Uurimistöö valmimise eest soovin kõige rohkem tänada juhendajat, Tartu Observatooriumi vanemteadurit, PhD Elmo Tempelit, kes pakkus välja huvitava uurimistöö teema ja vastas tekkinud küsimustele. Samuti soovin tänada koolipoolset juhendajat, füüsika- ja keemiaõpetajat Einike Reinvelti, kes juhendas töö kirjaliku osa vormistamist. Tänan ka uurimistöö seminaridel osalenud ning töö sisu kohta häid tähelepanekuid teinud õpilasi ja õpetajaid.

SISUKORD

SÕNASTIK

Andmekiht on rakenduses eraldi konfigureeritav ning sisse-välja lülitatav andmehulk.

Arenduskeskkond on komplekt tööriistu, mis muudavad tarkvaraarenduse lihtsamaks.

Arendustsükkel on pidev protsess, mille käigus valmib rakendus, mille hulka kuulub koodi kirjutamine, optimeerimine, testimine ja vigade parandamine.

Autowire on tööriist, mis ühendab rakenduse komponente automaatselt ning seega lihtsustab oluliselt programmi struktureerimist.

Galaktika on miljarditest tähtedest koosnev süsteem, mida hoiab koos gravitatsioon.

Galaktika filament on piirkond universumis, kus galaktikate tihedus on võrdlemisi suur.

Galaktika filamendi selgroog on galaktika filamenti iseloomustav mõtteline lõik.

Galaktikaparv on paljudest gravitatsiooniliselt seotud galaktikatest koosnev süsteem.

Galaktikate superparv on ruum, kus galaktikaparved liiguvad gravitatsiooni mõjul ühise tõmbepunkti poole.

Graafikakaart on arvuti komponent, mis kiirendab graafilise ruumi kujutamist.

Heliotsentriline universumi mudel on astronoomiline maailmapilt, mille järgselt on Päike terve universumi keskmes.

Jar-fail on käivitatav Java programm kokku pakitud kujul.

Kasutajaliides on vahend, mis võimaldab kasutajal hiire ja klaviatuuri abil rakendusega suhelda.

Kiildiagramm on joonis, mis kujutab universumi vaatlustulemuste läbilõiget.

Kommentaar koodis on koodi toimimist kirjeldav, kuid mitte mõjutav osa.

Kvasar on musta auku ümbritsevast ainest energiat saav ning väga eredalt kiirgav objekt.

Logi metaandmed on logiga kaasas käivad samuti olulised detailid, näiteks kuupäev, kellaaeg ja sõnumi tähtsustase.

Logimine on rakenduse toimingute arhiveerimine hilisemaks uurimiseks.

Lähtekood on rakenduse funktsionaalsust sisaldav, loetav ja muudetav koodikogu.

Must auk on kosmoseobjekt, millel on nii tugev gravitatsiooniväli, et sealt ei pääse isegi valgus välja.

Mustrisobitusalgoritm on mingi teksti mustri kirjeldus arvutile arusaadavas keeles, mille abil leiab arvuti sellised mustrid tekstist üles. Selle lihtsaim näide on tavaline tekstiotsing.

Objektorienteeritus on eelistus või suund programmeerimiskeeltes, mis võimaldab andmeid ja funktsioone iseseisvateks objektideks kokku grupeerida. Näiteks saab objekti luua igast galaktikast.

Optimeerimine on koodi ümberkirjutamine või arvuti jaoks operatsioonide arvu vähendamine, mille käigus programmi töö muutub kiiremaks tulemust oluliselt muutmata.

Parallaks on nurk kahest eri kohast ühte punkti suunatud kiirte vahel. Selle nurga järgi saab arvutada punkti kauguse vaatlejatest. Tähtede kauguse määramiseks vaadeldakse seda punkti pooleaastase vahega, kuna Maa liigub teisele poole Päikest.

Prototüüpimine on kiire, peamiselt ideede katsetamiseks mõeldud arendusprotsess, mil koodi puhtusele palju tähelepanu ei pöörata.

Puhas kood on kood, mille lugemine annab selge ülevaate programmi tööst ning ei tekita küsimusi.

Rakendus ehk programm on komplektne toode, mis sisaldab oma operatsioonide jaoks kõike vajalikku ning ei ole mõeldud osaliseks kasutamiseks mõne muu rakenduse sees.

Rakenduse ehitusprotsess on selle pakendamine, kasutades erinevaid selleks vajaminevaid tööriistu.

Rakenduse hangumine ehk selle kokku jooksmine on sündmus, mille käigus rakendus jääb liiga kauaks ühte käsku täitma, muutudes kasutuskõlbmatuks.

Teek on koodikogu, mis on mõeldud täielikuks või osaliseks kasutamiseks mõne teise rakenduse sees.

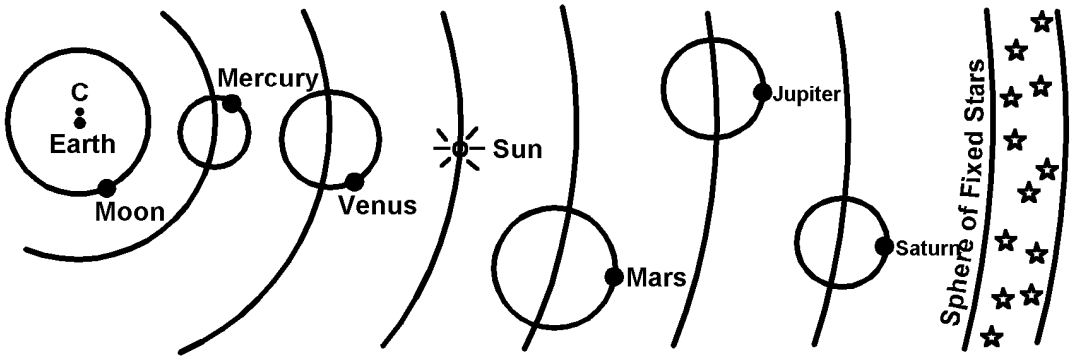
Tume aine on näiliselt massiivne aine, mis täidab gravitatsioonisimulatsioonide puudujääke.

Tume energia on näiline väli, millega kirjeldatakse gravitatsioonile vastupidist universumi kiirenevat paisumist.

1. UNIVERSUMI EHITUS

1.1. Ajalugu

Inimesed on proovinud pidevalt lahti seletada taeva olemust. Järelduseni, et Maa on ümmargune ja tiirleb ümber Päikese, jõuti juba antiikajal. Vana-Kreeka maadeavastajad nägid põhja või lõuna poole liikudes taevas erinevaid tähtkujusid. Pandi tähele, et kuuvarjutuse ajal oli vari ümar. Õpetlane Aristoteles käis välja idee, et taevakehad asuvad Maad ümbritsevatel kristallsfääridel. Ptolemaios asendas sfäärid orbiitide ja epitsüklitega – planeetide iseenesliku ringjoonelise liikumisega ümber punkti, mis omakorda liikus ümber Maa (joonis 1). Õpetlane Eratosthenes arvutas isegi Maa suuruse, uurides selleks varje erinevates asupaikades. Hilisematel ajastutel aga ei olnud võimalik eelkirjeldatud järeldusi kuidagi paremini tõestada ning samuti olid need vastuolus kristlike õpetustega, mistõttu jäi domineerima uskumus, et Maa on lapik ning on kogu maailma keskpunktiks. (Blackman: 2006)



Joonis 1. Ptolemaiose epitsüklitega universumimudel (Hawaii ülikool)

13. sajandil avastati uuesti Aristotelese sfääridega universumi mudel. Järgnes Poola astronoomi Mikołaj Koperniku vastuhakk kristlikule õpetusele 16. sajandi keskel. Ta leidis, et palju lihtsam on universumi keskmesse kinnitada Päike, kuna Aristotelese mudel oli liiga ebatäpne. Samuti täheldasid tähetargad, et vabalt liikuvad komeedid peaksid kristallsfäärid purustama. Tulemusena võeti kasutusele hoopis Ptolemaiose antiikne mudel.

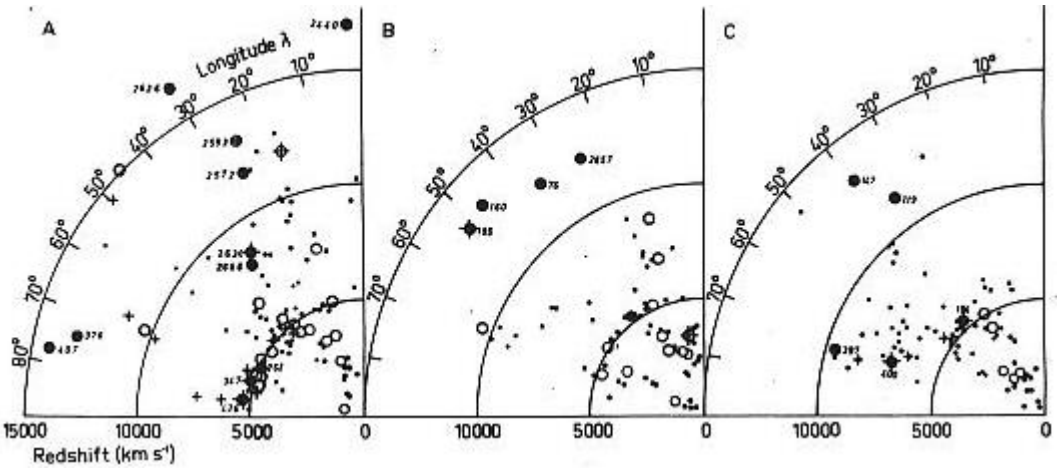
Alles 1609. aastal avaldas saksa matemaatik Johannes Kepler oma teooriad planeetide liikumise kohta ning kirjeldas Päikesesüsteemi mudelit. Samal aastal leiutas Itaalia astronoom

Galileo Galilei teleskoobi. See võimaldas lähemate planeetide uurimist, avastada nii kaaslasi, mis tiirlevad ümber lähemate planeetide, kui ka kaugemaid planeete. Sellel ajastul kinnistus heliotsentriline universumi mudel.

Saadi aru, et teised tähed on täpselt nagu meie Päike. Saksa matemaatik ja astronoom Friedrich Bessel suutis parallaksiga, taustanihke jälgimisega, ära mõõta mõnede tähtede kaugused maast. Parallaksi nihe on aga väga väike tähtedel, mis asuvad meist kaugel, mistõttu oli seda teha raske. (Tate: 2011)

1.2. Tänapäev

Teiste galaktikate olemasolu universumis avastas 20. sajandi esimesel poolel Eesti astronoom Ernst Julius Öpik (Tempel: 2010). Universumi kärjetaolist struktuuri täheldati esimest korda 1977. aastal, kui Eesti astronoom Mihkel Jõeveer kandis galaktikad kihiti koordinaatsüsteemi nii, et iga kiht iseloomustas selles kihis olevate galaktikate kaugust ja suunda Maast. Tekkinud kiildiagrammi (joonis 2) B-kihi keskel oli näha suurt tühimikku, mida uurides jõuti järeldusele, et universum näeb välja nagu mesilaskärg. Sellist mudelit peetakse senini õigeks. (Jaaniste: 1997–1999)



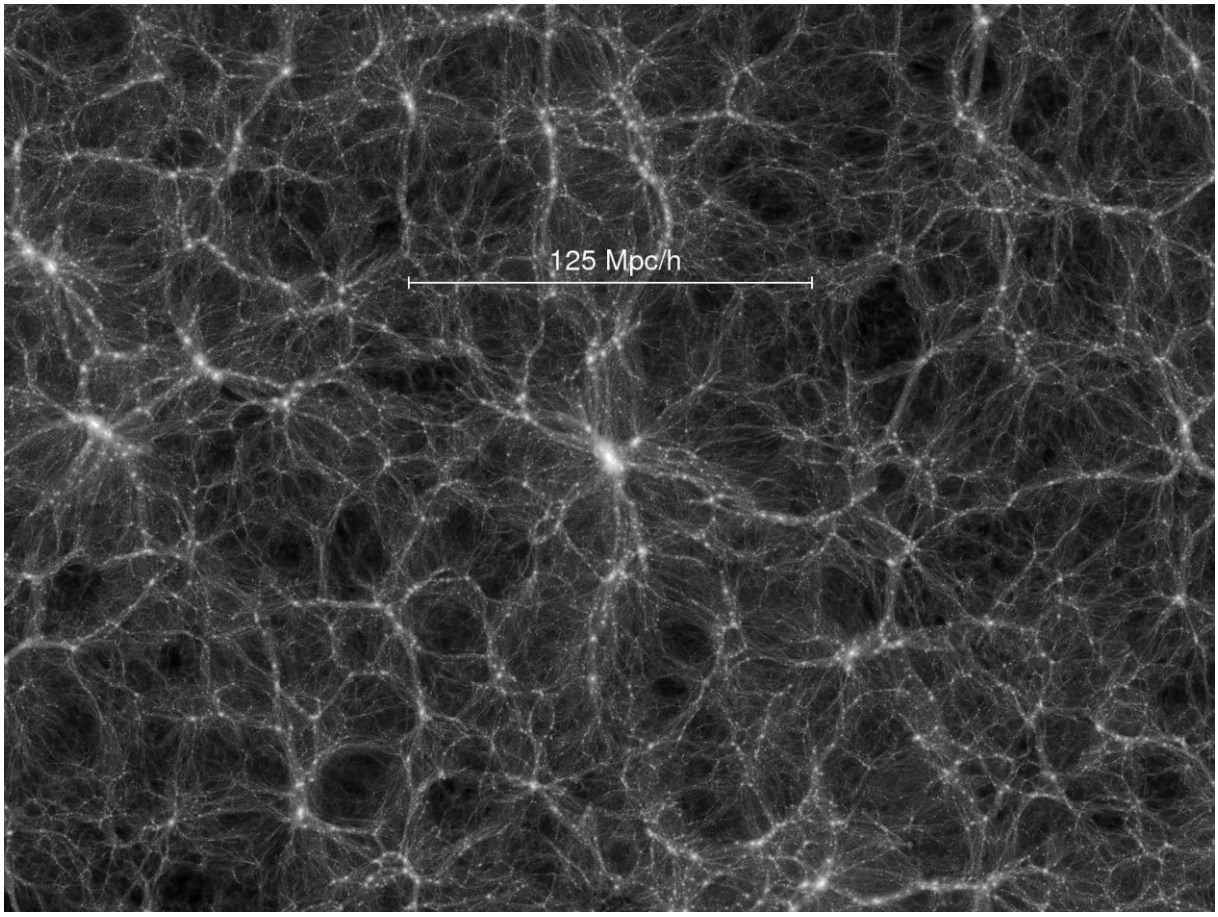
Joonis 2. kiildiagramm Perseuse tähtkuju suunas (M. Jõeveer, J. Einasto ja E. Tago, 1977)

Tänapäeva kosmoloogid näevad kosmost teistmoodi kui tavainimesed. Joonisel 3 on kujutatud universumi kärgstruktuuri. Seal on näha galaktikate filamentide, gravitatsiooni tõttu seotud galaktikate asetust. Kosmoses on üsnagi erinevaid kehi, millest mõned on nähtavad, mõned aga mitte. Nähtavate hulka kuuluvad tähed, gaas, tolm, galaktikad, kvasarid, galaktikaparved, superparved, aga ka reliktkiirgus ehk kosmiline mikrolaine-taustkiirgus. Kõiki neid on uuritud võrdlemisi kaua, kuna nende olemasolu on lihtne tõestada ja uurida taevast vaid teleskoobiga vaadeldes. Samuti on kosmoses nähtavat võimalik uurida nende värvi ja valguse intensiivsuse

järgi. Suuremaid ja kaugemaid nähtavate tähtede kogumeid grupeeritakse galaktikateks, galaktikaid omakorda galaktikaparvedeks.

Kosmoloogide arusaama järgi on kosmoses ka muud, mida me ei näe: tume aine, mustad augud ja tume energia. Neist tumeda aine ja tumeda energia olemasolu tõestavad puudujäägid arvutustes. Musti auke on võimalik tuvastada seetõttu, et aine liigub nende ümber iseäraliselt. Arvatakse ka, et reliktkiirgus võib anda informatsiooni universumi algse kuju kohta. (Tago: 2000)

Arusaam universumi ehitusest on aastasadade jooksul palju muutunud, kuna tehnoloogia on võimaldanud näha järjest kaugemale kosmosesse. Universumi kirjeldamine on siiski aga loogilise, hoomatava mudeli ehitamine faktidele ning nende hulk võib järgmiste aastasadade jooksul veel oluliselt kasvada. Seetõttu on tähtis luua selline tarkvara, mis suudaks toime tulla suurte andmemahtudega ning oleks uurijatele mugav, kergesti kättesaadav ja kasutatav.



Joonis 3. Universumi kärgstruktuur (Max Plancki Astrofüüsika instituut)

2. PROGRAMMEERIMINE

2.1. Arendustsükkel

Üldiselt osaleb arendusel kaks osapoolt, kelleks on tellija ehk klient ning arendaja. Klient esitab arendajale nõudmised, mida lõpp-produktis näha tahab. Arendaja vastutab rakenduse töökindluse ja nõudmistele vastavuse eest. Arendaja võib kliendile sealjuures soovitusi teha, kuid mõlemad osapooled peavad saavutama üksmeele rakenduse olemuses. Uurimistöö raames on kliendiks juhendaja Elmo Tempel ning arendajaks uurimistöö autor Gregor Eesmaa.

Arendustsükkel koosneb järgmistest osadest: kontseptsioon, algatus, analüüs, disain, valmistamine, testimine ja väljalase. Arendusprotsessi hõlbustamiseks on välja mõeldud erinevaid juhiseid. (Mikoluk: 2013)

Üks vanemaid ja tuntumaid neist on Waterfall'i ehk kose mudel. Selle järgi peab iga arendustsükli osa enne järgmise alustamist lõpetatud olema – justkui samm-sammuline, kosesarnane arendus. Selline lähenemine lihtsustab planeerimist, kuna nii arendaja kui ka klient peavad varakult põhjalikult läbi mõtlema, mida rakendus tegema hakkab. Samuti on töö maht juba alguses enam-vähem teada. Kose mudeli järgi loodud rakendustes võib esineda aga puudujääke, mida hiljem enam parandada ei tasu.

Selle vea parandamiseks on loodud Agile ehk paindlik mudel. See võimaldab rakenduse arenduse käigus muutusi palju lihtsamini läbi viia, kuna arendustsükli mitu osa võivad samal ajal kesta. Näiteks saab arendada ruttu välja algelise lahenduse, seejärel sinna funktsionaalsust juurde lisada. Samuti vajatakse pidevalt kliendi tagasisidet, mis annab võimaluse luua täpselt sellise rakenduse nagu klient soovib. Paindliku mudeli järgi käiv töö võib aga klienti tüüdata, samuti võivad muudatused põhjustada rakenduse pidevat ümberprogrammeerimist. Selliseks arendamiseks võib kuluda aega palju rohkem kui kose mudeli järgi. (Lotz: 2013)

Siinse uurimistöö praktilise töö arenduse jaoks valiti paindlik mudel, kuna see tagas pideva tagasiside kliendilt. Samuti nõudis see arendajalt tänapäevaste programmeerimistööriistade kasutusoskust, mida sai sel puhul õpitud. Sellise arendusviisi juures on aga vajalik arvestada, et rakenduse tööpõhimõte võib arenduse käigus muutuda.

2.2. Tööriistad

Klient esitas nõudmise, et rakendus peaks olema arendatud, kasutades Processing teeki Javale. Lisaks oli otsustatud kasutusele võtta paindlik arendusmudel. Seega sai valitud arenduseks järgnev tööriistade komplekt.

2.2.1. Java

Objektorienteeritud programmeerimiskeelt Java on väga hea kasutada kasutajaliidesega rakenduste jaoks. Esiteks on see väga paindlik, võimaldades prototüüpimist enne lõplikku rakenduse struktuuri paika panemist. Teiseks suudab Java erinevalt mõningatest teistest populaarsetest keeltest hallata oma mälukasutust ise, suutes ebavajalikud objektid mälust ise eemaldada. Kolmandaks töötab lähtekoodist masinale loetavaks tõlgitud rakendus mistahes operatsioonisüsteemi peal. (Cheng: 2004)

Lisaks Javale sisse ehitatud võimalustele on võimalik kasutada eraldi teeke. Teek on justkui programm, mis ei ole mõeldud töötama iseseisvalt. Selle asemel varustab ta teisi programme erinevate funktsioonidega, mida ühekordseks kasutuseks ise valmis programmeerida ja hallata oleks liiga suur töö.

2.2.1.1. Processing ja ControlP5

Processing on Java teek, mis võimaldab graafiliste lahenduste lihtsa arendamise. See kasutab arvuti graafikakaarti, et visualiseerida kolmedimensioonilist ruumi. Lisaks saab Processingu laiendusega ControlP5 luua kasutajaliidese. Tehniliselt pakub see erinevaid kasutajaliidese komponente, mis põhinevad Processingusse sisse ehitatud funktsioonidel – näiteks tavaline nupp konstrueeritakse sildist ja kastist ning sellel vajutamisele reageerimisest.

2.2.1.2. IDEA

Java programmide loomiseks on mõistlik kasutada arenduskeskkonda, mis automatiseerib rakenduse ehitamise ning võimaldab koodi ilusa ja puhta hoida. IntelliJ IDEA on Java arenduskeskkond, mis on ISIC õpilaspileti omanikele täiesti tasuta. See võimaldab kõikidele funktsioonidele ligipääsu klahvikombinatsioonidega, mida on mugav suurte projektide puhul kasutada. Samuti sisaldab IDEA erinevaid lisandmooduleid, mis võimaldavad kogu rakenduse arenduseks vajaminevaid tööriistu kasutada arenduskeskkonna siseselt.

2.2.1.3. Spring, Gradle

Spring on Java teek, mis lihtsustab programmeerimist mitmete erinevate võtetega, võimaldades näiteks eraldi märgitud objektide loomisel nende välju automaatselt täita. Samuti aitab Spring Java rakenduse muuta loogilisteks osadeks – komponentideks.

Gradle on tööriist, mis lihtsustab rakenduse ehitusprotsessi. See võimaldab teekide automaatse lisamise nime ja versiooni järgi. Lisaks veel mõningat ehitusautomaatikat, näiteks lihtsat jar-faili genereerimist. Ilma Gradle'ita tuleks teegid käsitsi alla laadida ja need omakorda käsitsi rakendusega ühendada.

2.2.1.4. Log4j, Exp4j, Guava

Arendatavas rakenduses kasutatakse mitmeid teeke, mis lihtsustavad olulisemate funktsioonide arendust. Ise nende funktsioonide arendamine võtaks liiga kaua aega ning ei pruugiks anda stabiilset tulemust.

Logimiseks ehk teadete väljastamiseks ning arendustöö hõlbustamiseks kasutati Log4j teeki. Log4j on logija, mis lisab teadetele metaandmeid, näiteks logiva protsessi nime ja kellaaja. Samuti võimaldab see logi eristamise tähtsuse järgi ning salvestamise eraldi faili.

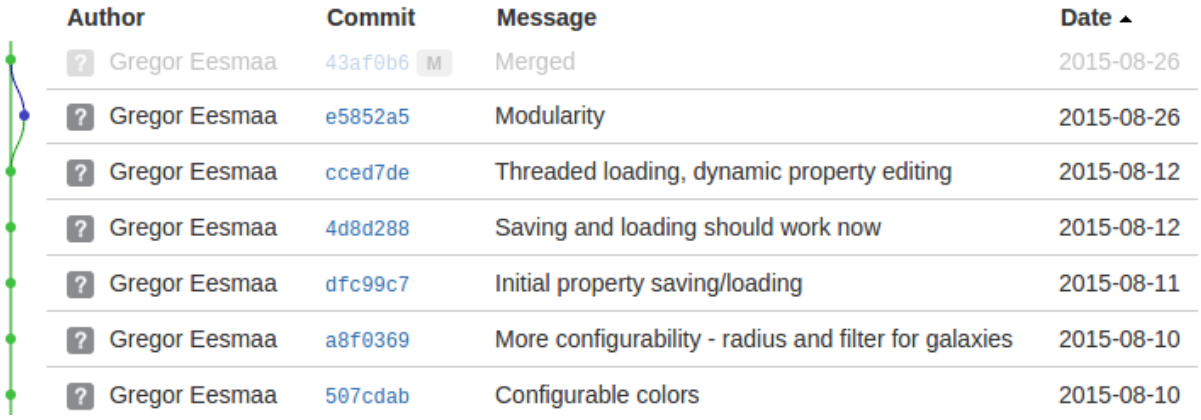
Exp4j on teek, mis võimaldab muutujatega valemitele väärtuste leidmise. Selle abil saab kasutaja poolt etteantud matemaatilist valemit rakendada failist loetud muutujate väärtuste korral. Visualiseerimisprogrammis läheb valemeid vaja objektide värvi määramiseks.

Guava on teek, mis pakub väga palju erinevaid lisafunktsioone, mida Javas sisse ehitatud ei ole. Selles uurimistöös kasutatakse Guava teegist funktsioone *little-endian* formaadis binaarfailide lugemiseks.

2.2.2. Git

Versioonihaldussüsteemide tööpõhimõte seisneb failidest perioodiliste tõmmiste pidamisel. Kui programmeerija on projekti faile muutnud, saab ta versioonihaldussüsteemi abil muudatused kommenteeritult registreerida ning hiljem neis navigeerida (näiteks uurida muutusi või taastada kindlaid tõmmiseid). Git on tuntud oma paindlikkuse poolest (Geduld: 2015). See võimaldab programmeerijal paralleelselt töötada mitmel harul (arendades mõlemal harul erinevat funktsiooni) ning need hiljem „kokku sulatada". Joonis 4 illustreerib tööd Gitiga.

Giti kasutamiseks on vaja keskserverit, kuhu muudatused saadetakse. See võib olla lokaalne ehk samas arvutis, kus arendustöö käib. Samuti võib kasutada tuntud teenuseid nagu GitHub või Bitbucket.



Author	Commit	Message	Date
Gregor Eesmaa	43af0b6 M	Merged	2015-08-26
Gregor Eesmaa	e5852a5	Modularity	2015-08-26
Gregor Eesmaa	cced7de	Threaded loading, dynamic property editing	2015-08-12
Gregor Eesmaa	4d8d288	Saving and loading should work now	2015-08-12
Gregor Eesmaa	dfc99c7	Initial property saving/loading	2015-08-11
Gregor Eesmaa	a8f0369	More configurability - radius and filter for galaxies	2015-08-10
Gregor Eesmaa	507cdab	Configurable colors	2015-08-10

Joonis 4. Töö Gitiga arendusprotsessi keskel

2.3. Kasutatavad programmeerimisvõtted

2.3.1. Kommentaarid

Peaaegu igas programmeerimiskeeles on võimalik kirjutada koodi sisse arendajapoolseid kommentaare nii, et need koodi käivitamist ei mõjutaks. Kommentaare kasutatakse koodi selgitamiseks lugejatele või ka iseendale meelespidamiseks. Javas loetakse kommentaariks tekst, mis kas algab sümbolitega "//” või on ümbritsetud sümbolitega “/*” alguses ja “*/” lõpus. Eristatakse veel ka dokumentatsiooni kommentaare, mis on ümbritsetud sümbolitega “/**” alguses ja “*/” lõpus, mida kasutatakse koodijuppide universaalseks kirjeldamiseks. Antud töös on kasutatud kommentaare koodinäidete põhjalikuks selgitamiseks.

2.3.2. Andmetüübid

Andmetüübid määravad ära, millist sorti väärtusega tegu on. Tänapäeva arvutid käsitlevad arvväärtusi baitidena, mis koosnevad kaheksast bitist. Bittidel on kaks võimalikku väärtust – kas 0 või 1. Järjestikusi bitte saab käsitleda binaararvuna ehk kahe astmete summana, milles arvestatakse ainult 1-väärtusega bittide järjekorranumbriga kahe astmeid (näiteks bittidele

1101 vastab arv $2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$). Seetõttu määravad andmetüübid ära ka võimalike väärtuste vahemiku. Programmeerimiskeeltes on kasutusel peaaegu universaalsed andmetüübid. Javas eristatakse järgmiseid andmetüüpe:

- *byte* – 1-baidine täisarvuline arvväärtus
- *short* – 2-baidine täisarvuline arvväärtus
- *int* – 4-baidine täisarvuline arvväärtus
- *long* – 8-baidine täisarvuline arvväärtus
- *float* – 4-baidine ratsionaalarvuline arvväärtus
- *double* – 8-baidine ratsionaalarvuline arvväärtus
- *char* – 2-baidine sümbol
- *boolean* – tõene või väär väärtus

2.3.3. *Big-endian* ja *little-endian* formaat

Andmetüüpide baite saab hoida kahte moodi. *Big-endian* formaadi järgi hoitakse kõige suuremaid kahe astmeid kirjeldavad baidid eespool ning väiksemaid kahe astmeid kirjeldavad baidid tagapool. *Little-endian* formaadi järgi aga hoitakse baite vastupidises järjekorras. Java *little-endian* formaadi lugemiseks tuge ei paku ning sellised väärtused tuleb programmis lugemisel ringi keerata.

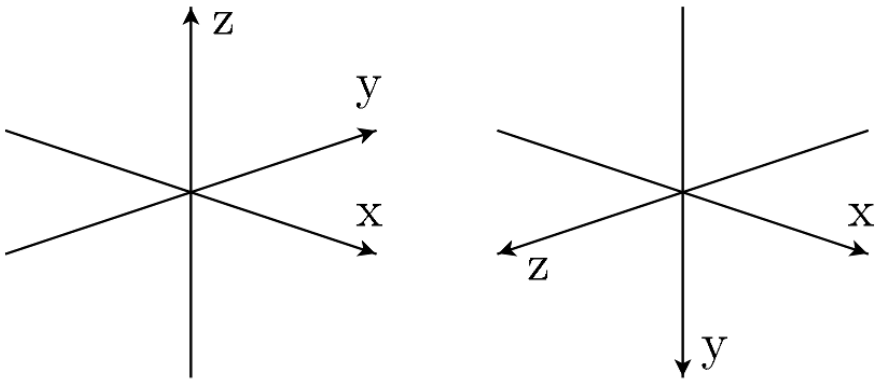
2.3.4. Regulaaravaldised

Regulaaravaldis ehk *regular expression* on tööriist, millega saab defineerida mustrisobitusalgoritme. Lihtsalt võib öelda, et sellega saab koostada teksti otsimise ja sobitamise valemeid. Antud uurimistöös kasutatakse seda vaid kõige lihtsamal kujul tekstifaili tükeldamiseks ükskõik kui paljude tühikute järgi valemiga “ +” (tühik ja pluss), mis kirjeldab mustrit, kus tühik esineb üks või mitu korda.

2.3.5. Y ja Z telje vahetus

Teadlaste ja programmeerijate koordinaatteljestike kasutus võib olla erinev. Reaalteadustes on tavaliselt koordinaatsüsteemis eestvaates X-telg suunatud paremale, Y-telg suunatud taha ning Z-telg suunatud üles. Kasutatava tarkvaraga on aga arvutigraafika ruumis eestvaates küll X-telg tihti suunatud paremale, kuid Y-telg alla ning Z-telg ette (joonis 5). Andmete kuvamiseks teaduslikult tuleb seetõttu arvutile y-koordinaadiks anda keha negatiivne z-koordinaat ning z-

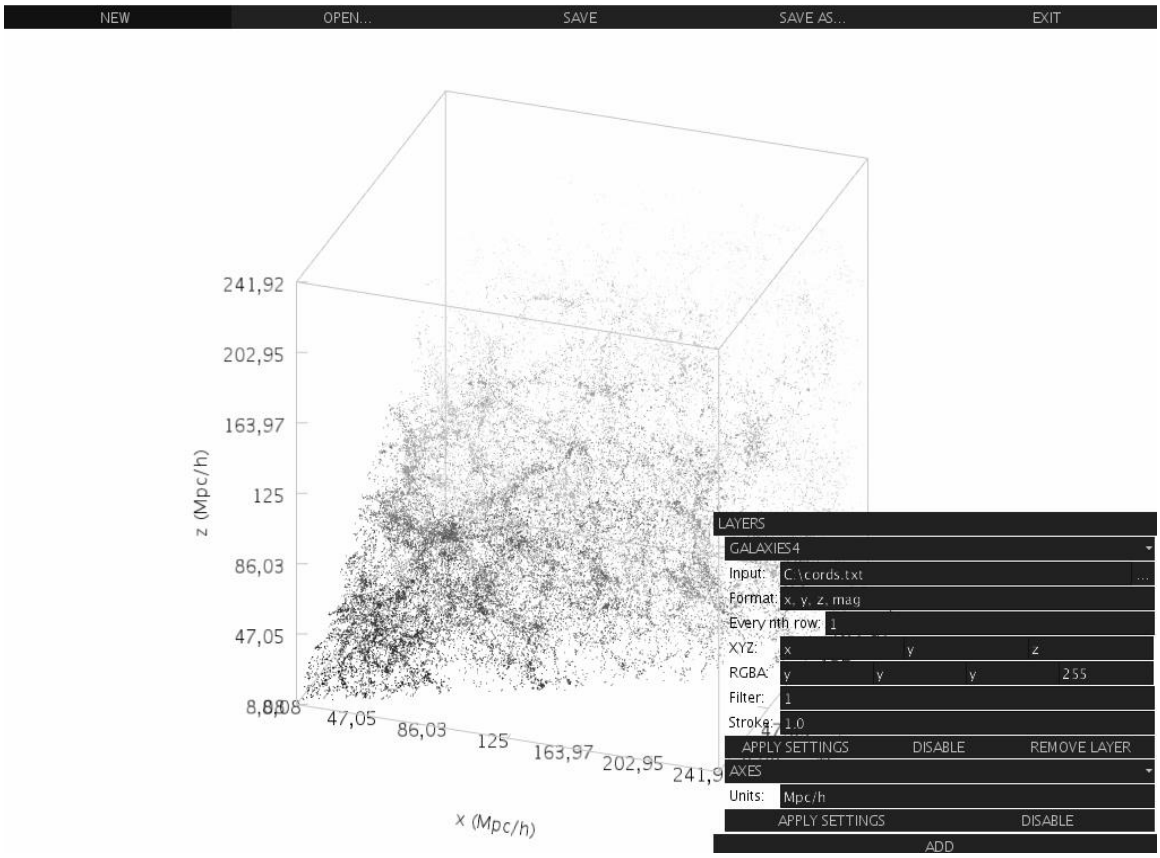
koordinaadiks negatiivne y-koordinaat. Selle tulemusena saab arvuti Y-teljest suunal alla Z-telg suunal üles ning Z-teljest suunal ette Y-telg suunal taha.



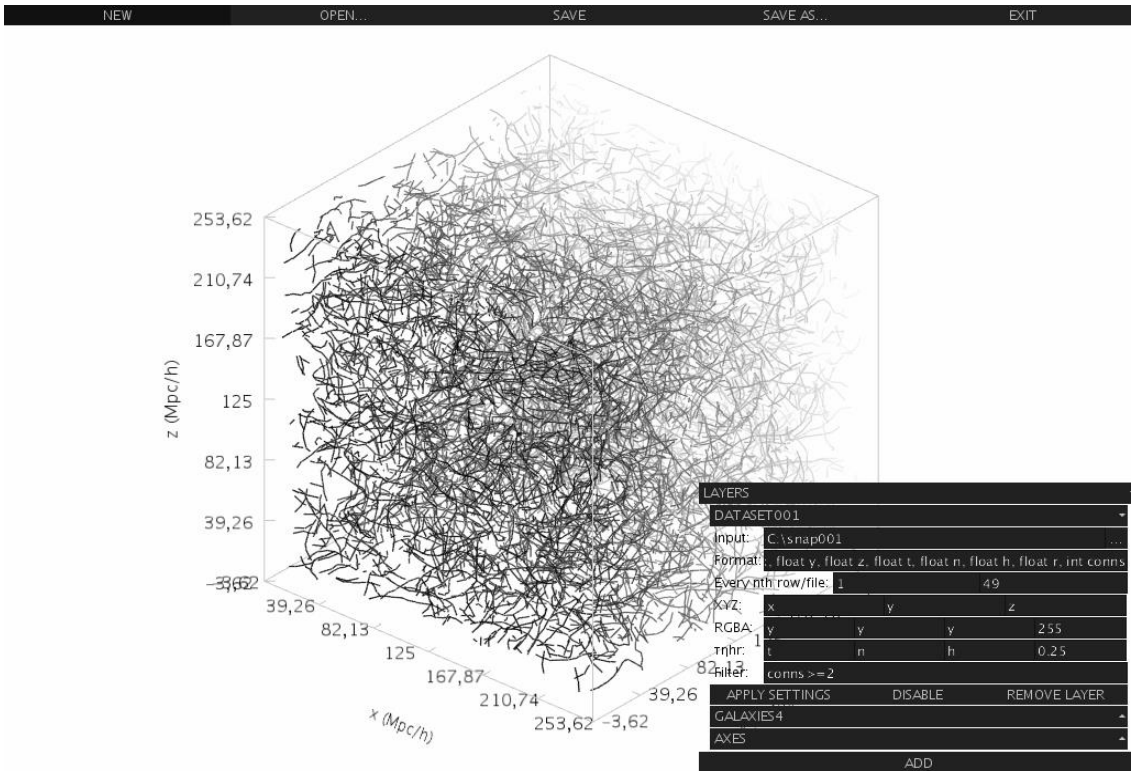
Joonis 5. Võrdlus teadusliku (vasakul) ja arvuti (paremal) koordinaatteljestiku vahel

3. TULEMUS

Praktilise töö raames valmis galaktikate ja galaktikate filamentide selgroogude (filamendi juppide) optimeeritud visualiseerimiskeskkond. Samuti valmis sellega kaasas käiv kasutusjuhend, kus on kirjeldatud, kuidas andmehulkade seadeid saab muuta. Joonistelt 6 ja 7 on näha, milline visualiseerimisprogramm välja näeb. Mõlemal joonisel on üks andmekiht, millel kujutatu värvikomponentide väärtused sõltuvad y-telje koordinaatidest, muutes 0-koordinaadile lähedama tumedamaks.



Joonis 6. Ekraanitõmmis valmis saanud programmist, kus on avatud suvaline galaktikate kiht.



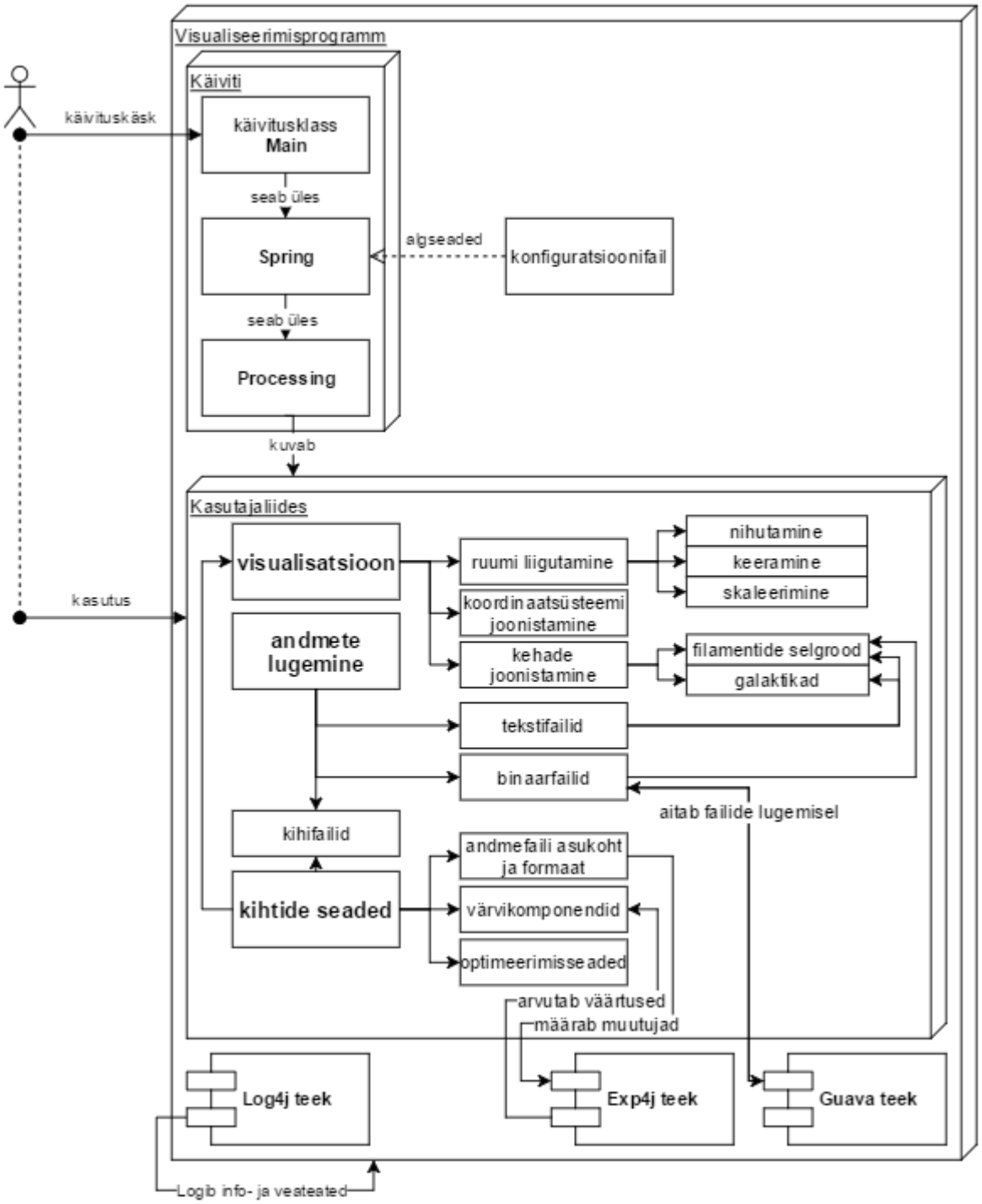
Joonis 7. Ekraanitõmmis valmis saanud programmist, kus avatud on suvaline filamentide selgroogude kiht.

3.1. Eesmärk

Loodud arvutiprogrammi eesmärk on visualiseerida kolmemõõtmelises ruumis etteantud tekstifailidest loetud galaktikaid, galaktikate filamentide selgrooge või binaarfailidest loetud galaktikate filamentide selgrooge. Programmiga peab olema võimalik näidata kihiti mitut erinevat andmehulka korraga. Lahti võetud andmekihte ja nende seadeid peab saama salvestada, et visualisatsiooni oleks võimalik korduvalt kasutada. Samuti peab programm suurte andmehulkadega sujuvalt töötama.

3.2. Programmi töö selgitus

Visualiseerimisprogrammi töö selgitamiseks on välja toodud tähtsamate protsesside töö koos kommentaaridega. Valminud programm tervikuna on palju mahukam ja täidab rohkem ülesandeid, kui toodud näited illustreerivad. Joonis 8 illustreerib programmi eri osade tööd ning teekide kasutust.



Joonis 8. Visualiseerimisprogrammi tööskeem

3.2.1. Andmekihid ja kihifailid

Andmekihtide näitamisega seotud seadeid saab muuta, samuti peab olema võimalus muudatused ka edaspidiseks kasutamiseks ära salvestada. Selleks on kasutusel Javasse

sisse ehitatud failitüüp Properties. Muuta saab andmefaili asukohta ning formaati, kihi tüüpi, värvikomponente ja optimeerimisseadeid.

Järgnevalt on toodud näide kihifailist, mis kirjeldab kolme eri tüüpi kihi visualiseerimise seadeid. Visualiseerimisprogrammi abil on võimalik sarnaseid faile avada, muuta ja salvestada.

axes.units=Mpc/h // Määrame teljestiku ühikud

dataset001.type=realisationslayer // Määrame uue kihi tüübi

dataset001.folder.path=C\:\\snap001 // Määrame kihi andmete asukoha

dataset001.optimisation.load_every_nth=10 // Määrame kihi „ülehüpatavate“ kehade arvu

dataset001.color.red=0 // Määrame kihi punase värvikomponendi väärtuse

dataset001.color.green=conns * 85 // Määrame kihi rohelise värvikomponendi valemi

dataset001.color.blue=0 // Määrame kihi sinise värvikomponendi väärtuse

dataset001.color.alpha=30 // Määrame kihi läbipaistvuse väärtuse

halos.type=objectslayer // Määrame uue kihi tüübi

halos.file.path=C\:\\halos_sn_1.txt // Määrame kihi andmete asukoha

halos.optimisation.load_every_nth=10 // Määrame kihi „ülehüpatavate“ kehade arvu

halos.color.blue=255 // Määrame kihi sinise värvikomponendi väärtuse

filamentbackbones2.type=filamentbackboneslayer // Määrame uue kihi tüübi

filamentbackbones2.file.path=C\:\\points.txt // Määrame kihi andmete asukoha

filamentbackbones2.file.format=idfil, npts, idpt, x, y, z, dx, dy, dz, vis, wvis, dstr, rad, ecode1, ecode2 // Määrame kihi andmete formaadi muutujad

filamentbackbones2.id=idfil // Määrame kihi grupeeringu muutuja nime

filamentbackbones2.color.blue=255 // Määrame kihi sinise värvikomp. väärtuse

```
rendersettings.transformation.translation.x=125 // Määrame algse keskpunkti x-koordinaadi

rendersettings.transformation.translation.y=125 // Määrame algse keskpunkti y-koordinaadi

rendersettings.transformation.translation.z=125 // Määrame algse keskpunkti z-koordinaadi
```

3.2.2. Käivitamine

Käsitletav programm kasutab nii Springi kui ka Processingu teeke, mistõttu on käivitamine keerulisem kui tavalise Java programmi puhul, milles ei ole vaja *Autowire*-töötlust ega graafikakaardiga joonistamist.

Järgnevalt on toodud kommenteeritud näide programmi käivitavast koodist. See võimaldab Springi ja Processingu teekide kasutamise terve programmi edasise töö jooksul.

```
package ee.tartu.jpg.gregor.gfv;


import ee.tartu.jpg.gregor.gfv.ui.UIApplet;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import processing.core.PApplet;


import javax.swing.*;


/**

 * Käivitab rakenduse, võimaldab Autowire-töötluse.

 *

 * Created by Gregor on 7/31/2015.
```

```
 */
public class Main extends Thread implements Runnable {


    @Autowired
    UIApplet uiApplet;


    /**
     * Käivitab Autowire-töödeldud rakenduse.
     */
    @Override
    public void run() {
        // Käivitame Processingu programmina juba olemasoleva Autowire-töödeldud
kasutajaliidese ilma argumentideta.
        PApplet.runSketch(new String[]{""}, uiApplet);
    }


    /**
     * Alustab rakenduse käivitamist.
     *
     * @param args ignoreeritakse.
     */
    public static void main(String[] args) {
        try {
            // Seame kasutajaliidese standardosade (nagu failidialoogi) väljanägemise.
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {
            // Vea korral logime veateate.
            e.printStackTrace();
```

```
        }

        // Loome uue põhja rakendusele, laadides failist Autowire seadistused.

        AbstractApplicationContext context = new ClassPathXmlApplicationContext("META-INF/spring/camel-context.xml");



        // Loome Autowire-töötlust võimaldava põhiklassi ning käivitame selle.

        Main main = context.getAutowireCapableBeanFactory().createBean(Main.class);

        main.run();

    }

}
```

3.2.3. Väärtuste automaatne seadmine

Springi teegiga kaasnev *Autowire* funktsioon võimaldab väljade lihtsa automaatse seadmise. Mõnel juhul läheb selleks vaja ka konfiguratsioonifaili, kus saab täpsustada algselt seatavaid väärtusi. Konfiguratsioonifaili on võimalik lihtsalt muuta ka pärast programmi kompileerimist.

Järgnevalt on toodud kommenteeritud näide automaatseid seadmisi kirjeldavast Springi konfiguratsioonifailist. Selles olev info loetakse programmi käivitamisel.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="[---]">

    <!-- Defineerime Springi komponentide otsimise koha. -->
    <context:component-scan base-package="ee.tartu.jpg.gregor.gfv"/>

    <!-- Defineerime UIAppleti loomisel seatavad väärtused. -->
    <bean id="UIApplet" class="*.gfv.ui.UIApplet">
        <!-- Defineerime kasutajaliidese interaktiivsuse komponendid, kus iga moodul vastutab ruumi mingisuguse transformeerimise eest. Ruumi saab pöörata, liigutada ja skaleerida. -->
        <property name="interactions">
            <list>
```

```
            <bean class="*.interactions.rotation.XRotationControl"></bean>
            <bean class="*.interactions.rotation.ZRotationControl"></bean>
            <bean class="*.interactions.rotation.YRotationControl"></bean>
            <bean class="*.interactions.translation.XTranslationControl"></bean>
            <bean class="*.interactions.translation.YTranslationControl"></bean>
            <bean class="*.interactions.translation.ZTranslationControl"></bean>
            <bean class="*.interactions.scaling.ZoomControl"></bean>
         </list>
      </property>
      <!-- Defineerime staatiliste aluskihtide komponendid – antud juhul ruumilise teljestiku.
-->
      <property name="staticLayers">
         <list>
            <bean class="*.layers.space.Axes"></bean>
         </list>
      </property>
   </bean>

   <!-- Määrame ära konteksti. -->
   <camelContext id="*.gfv" xmlns="http://camel.apache.org/schema/spring">
   </camelContext>

</beans>
```

Järgnevalt on toodud kommenteeritud näide koodi osast, mille väljade automaatset täitmist on kirjeldatud eelmises näites.

```
/**
* Ehitab Processing teegi abil kasutajaliidese
*
* Created by Gregor on 29.07.2015.
*/
@Component
public class UIApplet extends PApplet {


   /**
    * Logija.
    */
   private static final Logger LOG = LoggerFactory.getLogger(UIApplet.class);
```

```
    /**
     * Springi kontekst – võimaldab uute Autowire-töötlust võimaldavate objektide loomise.
     */
    @Autowired
    private ApplicationContext context;

    /**
     * Seaded, mida kasutatakse pildi joonistamiseks. Näiteks ruumi kaldenurk, nihe, suum.
     */
    @Autowired
    public RenderSettings rs;

    /**
     * Interaktiivsuse komponentide massiiv. See täidetakse automaatselt vastavalt Springi
     * konfiguratsioonifailile.
     */
    private Interaction[] interactions;

    /**
     * Staatiliste aluskihtide komponentide massiiv. See täidetakse automaatselt vastavalt
     * Springi konfiguratsioonifailile.
     */
    private Layer[] staticLayers;

    [---]
}
```

3.2.4. Failist andmete lugemine

Faile, millest andmeid loetakse, on kahte tüüpi. Ühed on tavalised tekstifailid, mida saab kasutada lihtsate andmete jaoks. Teised on binaarfailid, mida on vaja kettaruumi kokkuhoidmiseks suuremahuliste andmehulkade puhul – neid võib olla ka ühe andmekihi peale mitu, mistõttu laetakse terve kaust korraga. Binaarfailide puhul on tähtis ka andmetüüp, millest väärtuse baitide arv sõltub. Vastavalt kliendi nõuetele on binaarfailid *little-endian* formaadis.

Järgnevalt on toodud kommenteeritud näide tekstfaili lugemisest. Binaarfaile lugev kood on näitega sarnane, kuid keerulisem.

```
@Override
public void load() {
  super.load();
  // Eemaldame olemasolevad kehad.
  if (bodies == null)
     bodies = new ArrayList<>();
  else
     bodies.clear();


  // Kui faili ei valitud, siis ei tee midagi.
  if (filePath == null)
     return;
  try {
     // Loome faili lugeja.
     BufferedReader reader = new BufferedReader(new FileReader(filePath));
     String line;
     // Teavitame kasutajat faili lugemisest.
```

```
int prc = a.startProcess("Loading data for objects' layer \"" + name + "\"...");
LOG.info("Loading data for objects' layer \"" + name + "\"...");
try {
    // Tükeldame faili formaadi muutujateks, eraldajateks komad.
    String[] format = fileFormat.trim().split(", *");

    // Loome koordinaatide valemid, kasutades muutujaid seatud formaadist.
    Expression[] exps = new Expression[3];
    exps[0] = expb(x, format);
    exps[1] = expb(y, format);
    exps[2] = expb(z, format);

    // Loeme ja töötleme faili reakaupa, hüpates üle seatud arvuridade.
    int counter = 0;
    while ((line = reader.readLine()) != null) {
        if (line.startsWith("#") || ++counter < loadEveryNth)
            continue;
        counter = 0;

        // Tükeldame faili rea, eraldajateks tühikud (võib olla ka mitu).
        String[] row = line.trim().split(" +");
        // Loome taevakeha, seades muutujate väärtused.
        SpaceObject spaceObject = new SpaceObject(format, row);
        // Rakendame ettevalmistatud valemid koordinaatide arvutamiseks.
        spaceObject.load(exps);

        // Määrame uue piiri kõigile selle kihi kehadele.
```

```
            // See aitab hiljem reaalajas filtreerimisega – kui kihi ala jääb nähtava
            // ala sisse, ei ole vaja kehi transformeerimisel ükshaaval kontrollida.
            analyzeBody(spaceObject);

            // Lisame keha kihi.
            bodies.add(spaceObject);
        }
    } catch (IOException e) {
        // Püüame ja logime kõik veateated.
        LOG.error("Error parsing data", e);
    } finally {
        try {
            reader.close();
        } catch (IOException e) {
            // Püüame ja logime kõik veateated.
            LOG.error("Error closing file reader", e);
        }
    }
    // Anname kasutajale teada, et andmed on loetud.
    LOG.info("Loaded data for objects' layer \"" + name + "\"...");
    a.endProcess(prc);
  } catch (FileNotFoundException fnfe) {
    // Anname kasutajale teada, kui faili ei leitud.
    LOG.error("File " + new File(filePath).getAbsolutePath() + " does not exist.", fnfe);
    a.message("File " + new File(filePath).getAbsolutePath() + " does not exist.");
  }
}
```

3.2.5. Telgede joonistamine

Teljestiku joonistamisel tuleb tagada telgede siltide nähtavus iga nurga alt, mille tõttu keeravad sildid koos ruumi ja teljestiku pööramisega, olles alati loetavad. Samuti peab olema õige telgede skaala.

Järgnevalt on toodud kommenteeritud näide telgede joonistamisest. Telgede joonistamiseks vajalikud telgede asukohad arvutatakse enne joonistusprotsessi.

```
@Override
public void render(PGraphics g, boolean optimise) {
  // Seame teksti värviks musta.
  g.fill(0, 0, 0);
  // Seame joone värviks helehalli.
  g.stroke(200);

  // Alustame ruumi manipulatsiooni.
  g.pushMatrix();
  // Keerame ruumi vastavalt kasutaja seadetele.
  a.rs.applyRotation(g);
  // Joonistame nähtava ala kuubi.
  a.box(g, 2 * rasp);
  // Lõpetame ruumi manipulatsiooni.
  g.popMatrix();

  // Defineerime arvude formaadi reegli – 2 komakohta.
  DecimalFormat df = new DecimalFormat("#.##");
  // Joonistame X-telje skaala defineeritud arvujaotistega.
  for (int i = 0; i < xaxis.length; i++) {
    // Joonistame skaalajaotise.
```

```
        a.line(g, xaxisModel1[i], xaxisModel2[i]);
        // Arvutame, formaadime ja kirjutame skaala jaotise väärtuse.
        String str = df.format(xaxis[i].x / a.rs.getAsp() + a.rs.getDx());
        a.text(g, str, xaxisTxt[i].x - g.textWidth(str) / 2, xaxisTxt[i].y, xaxisTxt[i].z + 8);
    }
    // Joonistame Y-telje skaala defineeritud arvujaotistega.
    for (int i = 0; i < yaxis.length; i++) {
        // Joonistame skaalajaotise.
        a.line(g, yaxisModel1[i], yaxisModel2[i]);
        // Arvutame, formaadime ja kirjutame skaalajaotise väärtuse.
        String str = df.format(yaxis[i].y / a.rs.getAsp() + a.rs.getDy());
        a.text(g, str, yaxisTxt[i].x - g.textWidth(str) / 2, yaxisTxt[i].y, yaxisTxt[i].z + 8);
    }
    // Joonistame Z-telje skaala defineeritud arvujaotistega.
    for (int i = 0; i < zaxis.length; i++) {
        // Joonistame skaalajaotise.
        a.line(g, zaxisModel[i], zaxisModel[i].derive(lineLength, 0, 0));
        // Arvutame, formaadime ja kirjutame skaalajaotise väärtuse.
        String str = df.format(zaxis[i].z / a.rs.getAsp() + a.rs.getDz());
        a.text(g, str, zaxisModel[i].x - g.textWidth(str) - 10, zaxisModel[i].y, zaxisModel[i].z -
8);
    }

    // Joonistame X-teljele sildi ja ühikud.
    // Alustame ruumi manipulatsiooni.
    g.pushMatrix();
    // Nihutame keskpunkti X-telje sildi koordinaatidele.
    a.translate(g, x.x, x.y, x.z);
```

```
// Keerame ruumi vastavalt kasutaja seadetele.
a.rs.applyRotation(g);
// Pöörame silti ümber X-telje nii, et see liiguks kaameraga kaasa.
a.rotateX(g, a.rs.getRotationX() * (!xright ? 1 : -1) + a.PI * (!xright ? 1 : 0));
// Pöörame silti 180 kraadi, kui see asub vastasteljel, et see oleks õigetpidi.
a.rotateZ(g, a.PI * (!xright ? 1 : 0));
// Joonistame telje sildi ja ühikud (kui on seatud).
String str = "x" + (units == null || units.isEmpty() ? "" : (" (" + units + ")"));
g.text(str, -g.textWidth(str) / 2, rasp / 2.5f);
// Lõpetame ruumi manipulatsiooni.
g.popMatrix();


// Joonistame Y-teljele sildi ja ühikud.
// Alustame ruumi manipulatsiooni.
g.pushMatrix();
// Nihutame keskpunkti Y-telje sildi koordinaatidele.
a.translate(g, y.x, y.y, y.z);
// Keerame ruumi vastavalt kasutaja seadetele.
a.rs.applyRotation(g);
// Pöörame silti 90 kraadi ehk Y-telje sihti.
a.rotateX(g, a.HALF_PI);
// Pöörame silti ümber Y-telje nii, et see liiguks kaameraga kaasa.
a.rotateY(g, a.rs.getRotationX() * (!yright ? 1 : -1) + a.HALF_PI * (!yright ? 1 : -1));
// Pöörame silti 90 kraadi ehk Y-telje sihti.
// lisaks 180 kraadi, kui see asub vastasteljel, et see oleks õigetpidi.
a.rotateZ(g, a.HALF_PI + a.PI * (!yright ? 1 : 0));
// Joonistame telje sildi ja ühikud (kui on seatud).
```

```
  String str2 = "y" + (units == null || units.isEmpty() ? "" : (" (" + units + ")"));
  g.text(str2, -g.textWidth(str2) / 2, rasp / 2.5f);
  // Lõpetame ruumi manipulatsiooni.
  g.popMatrix();


  // Joonistame Z-teljele sildi ja ühikud.
  // Alustame ruumi manipulatsiooni.
  g.pushMatrix();
  // Nihutame keskpunkti Z-telje sildi koordinaatidele.
  a.translate(g, z.x, z.y, z.z);
  // Pöörame silti 180 kraadi suunale alt üles.
  a.rotateZ(g, -a.HALF_PI);
  // Joonistame telje sildi ja ühikud (kui on seatud).
  String str3 = "z" + (units == null || units.isEmpty() ? "" : (" (" + units + ")"));
  g.text(str3, -g.textWidth(str3) / 2, -rasp / 2.5f);
  // Lõpetame ruumi manipulatsiooni.
  g.popMatrix();
}
```

3.2.6. Hangumise vältimine

Suuremahuliste andmefailide visualiseerimisel ei ole mõttekas alati kõiki taevakehi näidata. Seetõttu kuvatakse ajal, kui kasutaja ruumi liigutab, ainult täpselt nii suur osa kõigist objektidest, et pildi liikumine inimsilmale sujuv tunduks. Kui kasutaja oma tegevuse lõpetab, joonistatakse pilt täielikult.

„Sujuv“ on programmile defineeritud kui tunde järgi valitud joonistussagedus 19–36 korda sekundis, ehk ühe pildi joonistamiseks lubatud aeg on 0,028–0,052 sekundit. Ajakulu vähendamiseks saab hüpata üle mitme joonistatava keha, mis tagab ka joonistatavate kehade

ühtlase jaotuse ruumis. Programm mõõdab piisavalt täpselt objektide joonistamiseks kuluvat aega ning kui see ei ole soovitud piirides, arvutatakse ülehüpatavate objektide arv uuesti.

```
@Override
public void render(PGraphics g, boolean optimise) {
  // Alustame ruumi manipulatsiooni.
  g.pushMatrix();
  // Keerame ruumi vastavalt kasutaja seadetele.
  a.rs.applyRotation(g);

  // Valmistame ette kiiresti ligipääsetava massiivi nähtavatest kehadest.
  prepVisibleBodies();
  // Kui optimeerimine on sisse lülitatud, hüppame üle joonistavate kehade, vastavalt
  // eelmisel hangumisel tehtud arvutusele.
  int rv = optimise ? showEveryNthOnInteraction : 1;
  // Määrame, kas üksiku keha joonistamisel võib kompromisse teha – tingimuseks on
  // 1000-st rohkem nähtavat keha.
  boolean efficient = optimise && visibleBodiesArr.length / rv > 1000;

  // Määrame joone paksuse vastavalt kasutaja seadetele.
  g.strokeWeight(strokeWeight);
  // Fikseerime ligikaudu nanosekundi täpsusega nähtavate kehade joonistamise algusaja.
  long startNanos = System.nanoTime();
  // Joonistame kõigist kehadest iga rv-nda keha.
  for (int i = 0; i < visibleBodiesArr.length; i += rv) {
     RenderableBody rb = visibleBodiesArr[i];
     if (rb.isVisible())
        rb.render(g, efficient);
```

```
  }
  // Fikseerime ligikaudu nanosekundi täpsusega nähtavate joonistamise lõpetamise aja.
  long endNanos = System.nanoTime();
  // Taastame joone paksuse.
  g.strokeWeight(1);


  // Kui optimeerimine on sisse lülitatud, arvutame vajadusel uue ülehüpatavate kehade
arvu.
  if (optimise) {
    // Leiame joonistamiseks kulunud aja sekundites.
    float renderTime = (endNanos - startNanos) / 1000000000f;
    // Leiame aja, mis peaks kuluma ühe kihi joonistamiseks, kui ühe korra kõikide
    // kihtide joonistamiseks peaks kuluma 0,0417 sekundit, mis on ligikaudu 24 korda
    // sekundis, mis peaks olema inimese silmale piisavalt sujuv.
    float attemptRenderTime = 0.0417f / a.getEnabledVisualizationLayerCount();


    // Muudame ülehüpatavate kehade arvu järgmise pildi joonistamisel ainult siis, kui
    // joonistamine oli soovitavast rohkem kui 1,25 korda aeglasem või rohkem kui 1,5
    // korda kiirem. Sellega väldime aja väikest kõikumist ja sellest tulenevat pildi
    // imelikku vilkumist. Kordajad valitud tunde järgi.
    if (renderTime>attemptRenderTime * 1.25F || renderTime<attemptRenderTime / 1.5F)
{
      // Teades kulunud aega (x), hüppe suurust (y) ja soovitavat aega (z), saame
      // arvutada uue soovitava hüppe suuruse (valemiga x * y / z).
      showEveryNthOnInteraction = (int) (renderTime * rv / attemptRenderTime);
      // Kui hüppe suurus on väiksem kui 1  ehk kui pilt joonistatakse liiga
      // kiiresti, nii et saaks veelgi rohkem kehi näidata, aga rohkem kehi ei ole
      // – olgu hüppe suurus minimaalselt 1.
```

```
        if (showEveryNthOnInteraction < 1)
            showEveryNthOnInteraction = 1;
      }
   }

   // Lõpetame ruumi manipuleerimise.
   g.popMatrix();
}
```

KOKKUVÕTE

Töö teoreetilises osas anti ülevaade universumi ehituse uurimise ajaloost. Praktilise töö raames valmis visualiseerimiskeskkond ning sellega kaasas käiv kasutusjuhend. Programm kirjutati keeles Java, kasutades eelkõige Processing teeki. Kasutatud tööriistad ja võtted on laialt levinud tänapäevases programmeerimismaailmas ning nende kasutamine andis hea töökogemuse ja ettekujutuse kaasaegsest arendustsüklist.

Valminud programm vastab kliendi ootustele: sellega on võimalik galaktikate ning galaktikate filamentide asukohti efektiivselt visualiseerida ning neid iga nurga alt vaadeldes uurida. Programmi kirjutades tuli peamiselt tähelepanu pöörata selle optimeerimisele ja kasutusmugavusele.

Kliendi ehk juhendaja Elmo Tempeli sõnul on ta rakendust kasutanud vaid mõnel korral, kuna on hiljem pigem teistsuguste teemadega tegelenud. Puuduseks toob ta välja, et pole saanud ligipääsu rakenduse lähtekoodile, et seda vajadusel ise edasi arendada või muuta.

Uurimistööst õppisin eelkõige tarkvaraprojektide planeerimist, arendust ning suuremahuliste andmehulkade optimaalselt visualiseerimist. Samuti mõistsin, et programmi lähtekoodi korrashoiule oleks tulnud pöörata suuremat tähelepanu, et seda oleks hiljem lihtsam kliendile esitleda või avalikustada. Suurimaks takistuseks osutus aga arendustööks kuluva aja

alahindamine – esialgu tundus, et etteantud ajavahemikus saan tehtud palju rohkem, kui tegelikult jõudsin.

ABSTRACT

Galaxies are not distributed evenly throughout the universe. They are gravitationally bound into a spiderweb-like structure. The best way to gain a better understanding of that structure is to visualize it by using computers. The aim of this research was to develop software that plots huge amounts of galaxies efficiently.

The software development was carried out using agile development model. Java was chosen as the programming language along with Processing library to utilize the graphics card for visualizations. Used tools and methods are common in modern software industry.

Finished software meets the customer's expectations: it can handle large amounts of data without any difficulty and helps to visualize galaxies and galaxy filaments by providing the user with complete control over how they are rendered. Most effort went into optimizing the program to work smoothly with large amounts of data, but also into improving user experience.

KASUTATUD KIRJANDUS

Blackman, Eric 2006. The Universe of Aristotle and Ptolemy. Astronomy 104 – The Solar System. http://www.pas.rochester.edu/~blackman/ast104/aristotle8.html. Vaadatud 29.03.2016.

Cheng, Betty H.C. 2004. Java for C/C++ Programmers. http://www.cse.msu.edu/~cse452/Fall2004/Lectures/10-java-part2.pdf. Vaadatud 06.12.2015.

Geduld, Marcus 2015. What is git and why should I use it? https://www.quora.com/What-is-git-and-why-should-I-use-it. Vaadatud 25.03.2016.

Jaaniste, Jaak 1997-1999. Universumi kärjetaoline struktuur. Kosmoloogia. Interaktiivne astronoomiaõpik. http://opik.obs.ee/osa4/ptk08/box01.html. Vaadatud 22.11.2015.

Lotz, Mary 2013. Waterfall vs. Agile: Which is the Right Development Methodology for Your Project? http://www.seguetech.com/blog/2013/07/05/waterfall-vs-agile-right-development-methodology. Vaadatud 25.03.2016.

Mikoluk, Kasia 2013. Agile Vs. Waterfall: Evaluating The Pros And Cons. https://blog.udemy.com/agile-vs-waterfall/. Vaadatud 25.03.2016.

Tago, Erik 2000. Universum aastatuhande vahetusel. I. – Vaatleja, nr 2. http://www.obs.ee/cgi-bin/w3-msql/vaatleja/artikkel.html?id=15. Vaadatud 22.11.2015.

Tate, Karl 2011. The History & Structure of the Universe (Infographic). http://www.space.com/13336-universe-history-structure-evolution-infographic.html. Vaadatud 11.12.2015.

Tempel jt = Tempel, Elmo, Radu Stefan Stoica, Vicent J. Martínez, Lauri Juhan Liivamägi, G. Castellan, Enn Saar 2013. Detecting filamentary pattern in the cosmic web: a catalogue of filaments for the SDSS. http://arxiv.org/abs/1308.2533. Vaadatud 22.11.2015.

Tempel, Elmo 2010. Kust on pärit Universumi ehituskivid? – Vaatleja, 17. dets. http://www.astronoomia.ee/vaatleja/3156/. Vaadatud 22.11.2015.