TARTU ÜLIKOOL
Arvutiteaduse instituut
informaatika õppekava

Tanel Orumaa

SOFIT Tasemeredaktor

Bakalaureusetöö (9 EAP)

Juhendajad:
Ulrich Norbisrath, PhD
Mark Muhhin, MSc
Peeter Nieler

Tartu 2022

SOFIT Tasemeredaktor

Lühikokkuvõte:

Selles töös on kirjeldatud Criffin OÜ-le loodud rakenduse SOFIT Tasemeredaktor arenduskäiku. Lõputöö käigus valmis minimaalne töötav rakendus, millega on võimalik ruumi plaan joonestada ning selle põhjal automaatselt kolmemõõtmeline võrestik genereerida. Rakendusele sooritati ka kasutajatestimine, millest selgus, et rakendust on piisavalt lihtne kasutada, sest esmakordselt seda kasutades said kõik kolm testijat keskmise raskusastmega hoone loomisega hakkama mõistliku aja jooksul.

Võtmesõnad:

Võrestiku genereerimine, kasutajamugavus, Delaunay triangulatsioon, varjutaja, programmeerimismustrid

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

SOFIT Level Editor

Abstract:

This thesis describes the implementation of the application SOFIT Level Editor which was created for Criffin OÜ. A minimum viable product was created, which allows the user to draw a blueprint of a building and automatically generate a 3D mesh from it. A usability testing scenario was also performed for the application. The result was that the application is simple enough to use that all three of the testers managed to create a medium difficulty building within a reasonable time when using the application for the first time.

Keywords:

Mesh generation, usability testing, Delaunay triangulation, shader, programming patterns

CERCS: P170 Computer science, numerical analysis, systems, control

Sisukord

1. Sissejuhatus 4
2. Sarnased rakendused 6
2.1. Sweet Home 3D 6
2.2. Unity Probuilder 7
2.3. Blender 7
3. Implementatsioon 9
3.1. Kasutatud tehnoloogiad 10
3.2. Hoone plaani joonestamise režiim 11
3.2.1. Ruudustik 11
3.2.2. Plaani joonestamine 13
3.3. Kolmemõõtmelise võrestiku loomine 15
3.3.1. Ettevalmistused võrestamiseks 16
3.3.2. Seinte võrestike genereerimine 18
3.3.3. Põrandate ja lagede võrestike genereerimine 23
3.4. Kasutatud programmeerimismustrid 33
3.4.1. Olekumuster 34
3.4.2. Singleton'i muster 34
3.4.3. Teenuseotsija muster 35
3.4.4. Jälgija muster 35
3.4.5. Käsumuster 36
3.5. Kasutajamugavus 37
3.5.1. Tähelepanekud CAD rakenduste kasutajaliidestes 38
3.5.2. Head tavad kasutajaliideseid disainides 40
4. Kasutajatestimine 43
4.1. Testijad 43
4.2. Testi stsenaarium 43
4.3. Testimise tulemused 44
5. Kokkuvõte 47
Kasutatud kirjandus 48
Lisad 49
A. Tööga seotud failid 49
B. Kasutajatestimisega seotud failid 50
C. Litsents 51

1. Sissejuhatus

Criffin[1] on 2012. aastal asutatud ettevõte, mille tegevus on suunatud virtuaalreaalsusega seotud tehnoloogiatele. Ettevõte tegeleb teadustöö ja arendusega, et kaotada vahe päris elu ja virtuaalreaalsuse vahel. Nad on arendanud mitmeid omnisuunalisi jooksulinte[2], juhtmevabasid jälgimissüsteeme ning teisi seadmeid, mis aitavad virtuaalreaalsust elutruumaks teha. Lisaks arendavad nad ka erinevaid tarkvaralisi lahendusi, mis eelnevalt mainitud seadmeid kasutades suudavad inimkeha virtuaalselt simuleerida ja jäsemete täpseid liikumisi rakenduses peegeldada.

Üks sellistest rakendustest on Unity[3] mänguarendusplatvormil arendatav tarkvara SOFIT („Special Operating Forces Immersive Training”). Selle rakenduse eesmärk on pakkuda turvalist treeningkeskkonda erinevate eriolukordade treenimiseks kasutades virtuaalreaalsuse peakomplekte. Rakenduses on võimalik koostöös harjutada näiteks pantvangikriisi lahendamist või hoone vastastest puhastamist linnalahingu olukorras. Rakenduse põhisihtgrupp on hetkel sõjavägi ja politsei eriüksused, kuid tulevikus on plaanis laieneda ka teistele sihtgruppidele.

Criffin soovib luua eraldiseisvat rakendust, millega on lõppkasutajatel võimalik luua stsenaariumeid tarkvarale SOFIT. Selle rakendusega peaks saama luua mängumaailma osad (esmajärgus hooned), paigutada maailma objekte, seada üles stsenaariumi tingimused ja reeglid ning lõpuks ka kogu loodud stsenaariumi eksportimine rakendusse SOFIT. Terve sellise rakenduse arendus on väga mahukas töö ning seetõttu valiti käesoleva töö skoobiks luua minimaalselt töötav rakendus, millega saaks luua lihtsakoelisi mängumaailmu (hooneid) ning paigutada objekte maailma.

Lõputöö käigus valmis rakendus SOFIT Tasemeredaktor (ingl SOFIT Level Editor), millega saab kasutaja märkida plaanile loodava hoone plaani (seinad, uksed ja aknad), millest rakendus genereerib 3D mudeli. Sellesse mudelisse on kasutajal võimalik paigutada eelnevalt loodud objekte, näiteks mööbliesemeid või vastaseid, kusjuures need objektid on valmis mänguüksused, ehk neil on küljes juba vajalikud komponendid ja skriptid. See tähendab, et kasutaja jaoks on vajalik vaid objekt mängumaailma lisada ja rohkem nende pärast muretsema

[1] https://criffin.com/about-criffin/
[2] Jooksulindi laadne seade, millega on võimalik liikuda igas suunas (360°). Tüüpiliselt kasutatakse selliseid seadmeid koos virtuaalreaalsuse peakomplektidega, mis võimaldab lõpmatut liikumist mängumaailmas, olles reaalses maailmas koha peal.
[3] https://unity.com/

ei pea. Kõik 12 objekti, mida valminud rakenduses mängumaailma lisada saab, varustas lõputöö jaoks Criffin ning töö autor lisas rakendusse vaid funktsionaalsuse nende kettalt lugemiseks ja mängumaailma paigutamiseks.

Tasub veel välja tuua, et Criffini poolelt toetasid lõputöö valmimist Peeter Nieler, kes käitus põhiliselt kliendina ning Ingmar Trump, kes oli arendustugi. Kord nädalas (jaanuarist maini) toimusid ka arenduskoosolekud Criffiniga, kus lõputöö autor näitas eelmise nädala progressi ette ja sai esmast tagasisidet, soovitusi ning sai vajadusel nõu küsida.

Viimaseks tähtsaks märkuseks on see, et lõputöö raames arendatud rakendus ei ole vabavaraline. Enne arenduse algust kirjutas lõputöö autor alla lepingule, mis sätestab, et kõik töö käigus kirjutatud kood on Criffin OÜ omand ning töö autor seda pärast lõputöö esitamist levitada ei tohi. Seetõttu ei ole ka lõputööga kaasas rakenduse koodi. Juhul, kui on siiski mingil põhjusel vaja ligipääsu koodibaasile (näiteks lõputöö retsenseerimisel), võtta ühendust lõputöö autoriga emailil orumaa.tanel@gmail.com.

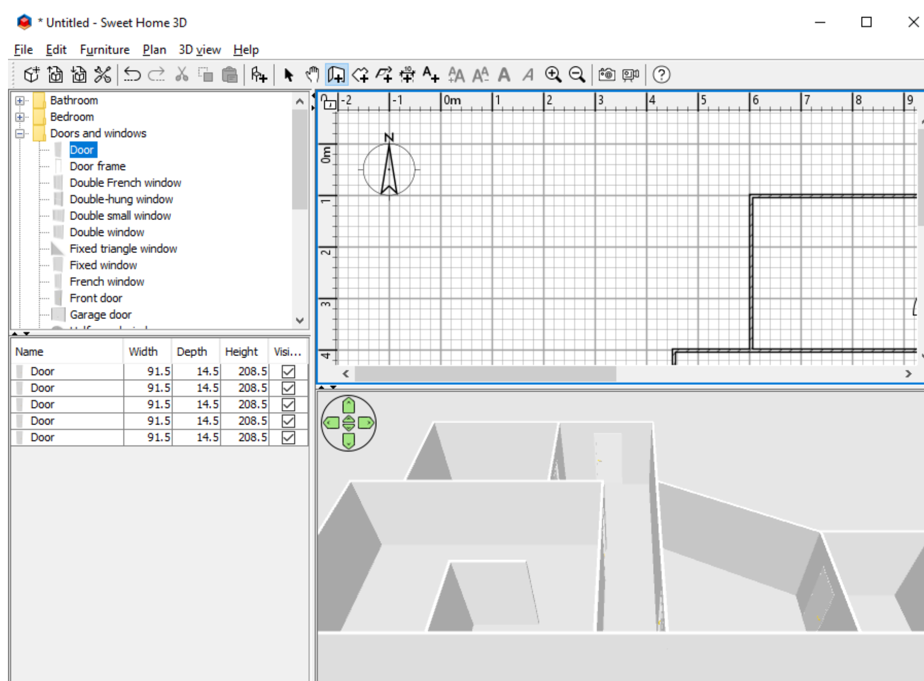
Peatükis 2 tutvustatakse olemasolevaid rakendusi, millest lõputöö autor inspiratsiooni sai. Kolmandas peatükis on lõputöö põhiosa – rakenduse implementatsiooni kirjeldus. Seal on välja toodud nii keerulisemate rakenduse osade arenduse kirjeldus kui ka tehtud valikud rakenduse arendamisel. Neljas peatükk räägib põgusalt kasutajatestimisest ja selle põhjal tehtud järeldustest. Viies peatükk toob välja kokkuvõtte tehtud tööst. Esimeses lisas on välja toodud tööga seotud failid (vt Lisa A: Tööga seotud failid) ja teises lisas kasutajatestimisega seotud failid (vt Lisa B: Kasutajatestimisega seotud failid).

2. Sarnased rakendused

Enne uue rakenduse looma hakkamist peaks tutvuma juba olemasolevate rakendustega. Seda kahel põhjusel: veendumaks, et ei eksisteeri juba rakendus, mis vajaliku eesmärgi täidaks ning et saada inspiratsiooni uue rakenduse loomisel. Rakendusi, mida saab kasutada mängumaailmade loomiseks on palju, kuid lõputöö autor tõi välja neist kolm: Sweet Home 3D[4], Unity Probuilder[5] ja Blender[6]. Neist esimese suure sarnasuse tõttu loodava rakendusega ning teised kaks seetõttu, et need on seotud mängutööstusega ning nendega on autoril olnud suurim kokkupuude. Allpool on lühidalt kirjeldatud kolme mainitud rakendust ja on toodud välja nende positiivsed ja negatiivsed küljed.

2.1. Sweet Home 3D

Sweet Home 3D on vabavaraline sisekujunduse rakendus, mille esimene versioon avalikustati juba 2006. aastal. Rakendust saab kasutada operatsioonisüsteemidel Windows, Linux ja MacOS. 2008. aastast alates on olemas ka veebiversioon WebGL[7] toega brauseritele.



Joonis 1. Kuvatõmmis rakendusest Sweet Home 3D

[4] https://www.sweethome3d.com/
[5] https://unity.com/features/probuilder
[6] https://www.blender.org/
[7] Web Graphics Library ehk WebGL on JavaScriptis kirjutatud liides 2D ja 3D graafiliste elementide renderdamiseks brauseris (https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API).

Rakenduse ülesehitus on sarnane loodava rakendusega – seal saab joonestada hoone plaani, mille põhjal genereeritakse kohe ka 3D mudel hoonest (vt Joonis 1). Lisaks saab loodud hoonesse paigutada mööbliesemeid. Kuna Sweet Home 3D rakendust on väga pikalt arendatud, on see väga võimekas. Samas on selle kasutamine üsna keeruline, sest tegemist on eelkõige siseplaneerimiseks, mitte mängumaailmade loomiseks mõeldud rakendusega ja läbi aja on rakendusele palju funktsionaalsust lisatud.

Teine suur piirang on see, et loodud hoonesse saab paigutada vaid mudeleid, mitte valmis mänguüksuseid. Seega ei saa tervet mängumaailma ühes rakenduses valmis teha, vaid peaks kasutama veel kolmandat rakendust, mis maailmas olevatele objektidele lisaks külge ka funktsionaalsused. Olenemata välja toodud piirangutest sai lõputöö autor siiski sellest rakendusest üsna palju inspiratsiooni.

2.2. Unity Probuilder

Erinevalt Sweet Home 3D rakendusest on Unity Probuilder plugin mõeldudki mängumaailmade prototüüpimiseks ja loomiseks. Sellega saab luua geomeetrilisi objekte, lisada päästikutsoone (alad, kuhu sisenedes midagi juhtub) ja arvutada automaatselt navigatsioonialasid (alad, kus eelprogrammeeritud mänguobjektid liikuda saavad).

Selle plugina suurimaks plussiks on see, et loodud maailmad on juba Unity mänguobjektid, seega oleks nende importimine SOFIT rakendusse lihtne. Samas pole selles hoonete loomine kuidagi suunatud – kasutaja peab ise jälgima, millised geomeetrilised objektid ta loob. Kuigi see annab kasutajale võimaluse luua ükskõik mida, lisab see samas keerukust tavaliste hoonete loomisel.

Miinuseks on veel see, et Unity Probuilder'i kasutamiseks peab olema installeeritud ka Unity ise. Võrreldes SOFIT Tasemeredaktor rakendusega, mis võtab kettal ruumi natuke üle 100 MB, võtab Unity 2021.2.8f1 versioon lõputöö autori arvuti kettal ruumi ligi 5.5 GB.

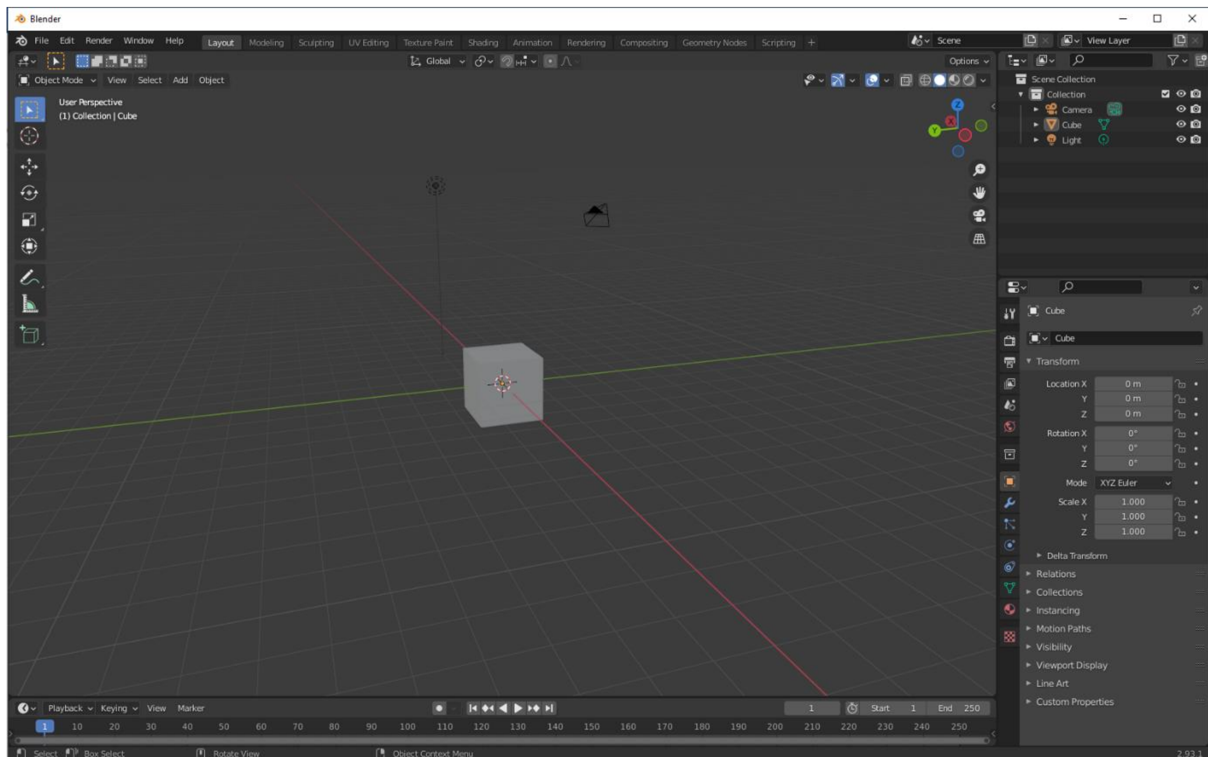
Nagu eelmisel rakendusel, on ka siin probleemiks liiga suur funktsionaalsuste hulk. Probuilder ise pole väga keeruline, kuid seda sisaldav Unity mängumootor on seevastu väga võimekas ja selle kasutamaõppimine võtab üsna pika aja.

2.3. Blender

Sarnaselt Unity Probuilder pluginaga on ka Blender seotud mänguarendusega. Tegemist on vabavaralise rakendusega, millega saab luua 3D mudeleid, neid animeerida, ning mitmeid muid

modelleerimisega seotud tegevusi teha. Sellel on tugi opertsioonisüsteemidele Windows, Linux ja MacOS.

Blenderis loodud faile on võimalik importida otse Unity'sse, mis teeb selle kasutamise koos Unity platvormiga mugavaks. Samas ei ole see rakendus sobiv SOFIT rakendusele mängumaailmade loomiseks. Üks põhjus on selles, et nagu ka Sweet Home 3D rakenduses, luuakse siin vaid mudeleid, mitte valmis mängumaailmu. Sarnaselt saaks need mudelid kolmandas rakenduses sobivaks muuta, kuid see lisaks ainult keerukust kogu protsessile.



Joonis 2. Kuvatõmmis rakendusest Blender.

Kuna Blender on kõikidest mainitud rakendustest kõige võimekam[8], on seda ka kõige keerulisem kasutada. Ka jooniselt (vt Joonis 2) on näha kümned menüükaustad, kuhu võimalikud funktsionaalsused grupeeritud on ning menüüde hulk vaid kasvab kui võtta arvesse ka rakenduse teisi režiime.

Lõputöö autorile aga Blender'i kasutamine meeldib, sest kui seda kasutama õppida, saab väga keerulisi tegevusi teha väga kiiresti. Seda suures osas seetõttu, et peaaegu kõik tihedamini kasutatavaid funktsionaalsuseid saab aktiveerida kiirklahvidega. Lisaks on Blender'i kasutajaliides olenemata suurest keerukusest hästi implementeeritud. See näeb ilus välja ning pärast mõningast rakendusega tutvumist on menüüde hierarhia intuitiivne.

[8] Nimekirjaga funktsionaalsustest saab tutvuda siin: https://www.blender.org/features/

3. Implementatsioon

Arenduse eelsel vestlusel Criffiniga selgusid mõned nõuded rakenduse implementatsiooni kohta. Üheks nõudeks oli rakenduse kasutusmugavus – rakendust peab olema lihtne kasutada. Lõppkasutajate arvuti kasutamise oskus ei pruugi olla väga hea ja see ei tohiks tasemete loomisel takistuseks saada.

Teine nõue oli, et loodud maailmad peaksid olema visuaalselt võimalikult realistlikud, kuid samas hästi optimeeritud, sest SOFIT peab virtuaalreaalsuse peakomplektidel töötama kõrge kaadrisagedusega. Teist osa sellest nõudest ei ole võimalik rakenduse nii varases etapis kontrollida, kuid sellegipoolest pööras lõputöö autor tähelepanu ka optimeerimisele.

Ülejäänud implementatsioonivalikud lubas Criffin lõputöö autoril teha. Rakenduse skoop oli samuti lahtine, kuna nii lõputöö autor kui ka Criffini esindajad ei osanud hinnata projekti alguses tööle kuluvat aega. Seda ka seetõttu, et täpsed nõuded polnud ette määratud, vaid tekkisid ja muutusid arenduse käigus.

Rakenduses on kolm põhilist osa ning üks toetav osa. Põhilistest osadest kaks – hoone plaani joonestamise režiim ning objektide paigutamise režiim – on kasutajaliidesega ja vajavad kasutaja sisendit oma tegevusteks. Viimane põhiline osa – kasutaja joonestatud plaanist 3D võrestiku genereerimine – toimub automaatselt pärast hoone plaani joonestamist. Ülalmainitud toetav osa on kaamera manipuleerimine, mis toimub sarnaselt üle terve rakenduse.

Selles peatükis on kirjeldatud rakenduse arenduskäik, selle käigus tekkinud probleemid ning kasutatud lahendused. Peatükk on jagatud viieks alapeatükiks, millest esimeses on toodud välja kasutatud tehnoloogiad ja põhjused, miks osutusid valituks just need. Teises alapeatükis on kirjeldatud hoone plaani joonestamise arendust ja natuke vastavat äriloogikat. Seejärel on välja toodud tehnoloogiliselt kõige keerulisem osa rakendusest – 3D võrestiku genereerimine. Neljas alapeatükk on pühendatud kasutatud programmeerimismustritele ja viimases peatükis on kirjeldatud kasutajaliidese disainis tehtud valikud ja nende põhjendused. Objektide paigutamise režiimi ja kaamera manipuleerimise implementatsioonidest töö autor täpsemalt ei räägi, kuna võrreldes teiste funktsionaalsustega oli neid üsna lihtne implementeerida ja erilisi takistusi ega huvitavaid lahendusi nendega seoses ei esinenud.

3.1. Kasutatud tehnoloogiad

Mänguarendusplatvormi valiku tegi Criffin. Kuna SOFIT rakendust arendatakse Unity mängumootorile, siis on kliendi soov, et sama tehtaks ka SOFIT Tasemeredaktor rakendusega. Üks põhjus selleks oli see, et lõputöö käigus valmiv rakendus on vaid minimaalselt töötav toode (MVP – „minimum viable product”) ja selle arendusega jätkab Criffin, kellel on juba olemasolev Unity kompetents. Lisaks oli Criffinil lihtsam tehtud tööd kontrollida ja tuge pakkuda mängumootoril, millega nad on kõige paremini tuttavad. Kolmas põhjus, mis pole küll lõputöö skoobis, on see, et tulevikus on plaan loodud tasemeid salvestada Unity stseeni (ingl „scene”) objektidena, et neid mugavalt põhimängu importida ja seda poleks mõnel teisel mängumootoril (vähemalt suure lisatööta) võimalik teha. Lõputöö autoril oli projekti alustades juba varasem kogemus Unity platvormiga olemas, seega sobis talle valitud mängumootor.

Kuna mängumootori valikuks oli Unity, siis oli sellega valitud ka kasutatav programmeerimiskeel – C#[9]. Varasemates Unity versioonides oli võimalik valida arenduskeeleks ka UnityScript (JavaScripti-laadne programmeerimiskeel), aga alates 2017.2 versioonist alates seda enam ei toetata[10]. C# on Microsofti arendatud objektorienteeritud programmeerimiskeel, millel on ka modernse programmeerimiskeele funktsionaalsused, nagu näiteks lambda avaldised ja LINQ (Language Integrated Query) süntaksi võimekus (sarnane Java striimidega).

Ka versioonihaldustarkvara valik tehti kliendi poolt. Projekt oli üles seatud GitLab[11] keskkonda, mis kasutab versioonihalduseks vabavaralist Git[12] tarkvara. Projekti hoidla (ingl. „repository”) ja Unity sätted seadis üles Ingmar Trump Criffinist enne projekti algust.

Kasutatud tehnoloogiana võib välja tuua veel JetBrains Rider[13] integreeritud programmeerimiskeskkonda (ingl „integrated development environment”), mis oli abiks projekti koodi kirjutamisel ja mille võimekas silumise tugi aitas mitmeid keerulisi probleeme lahendada.

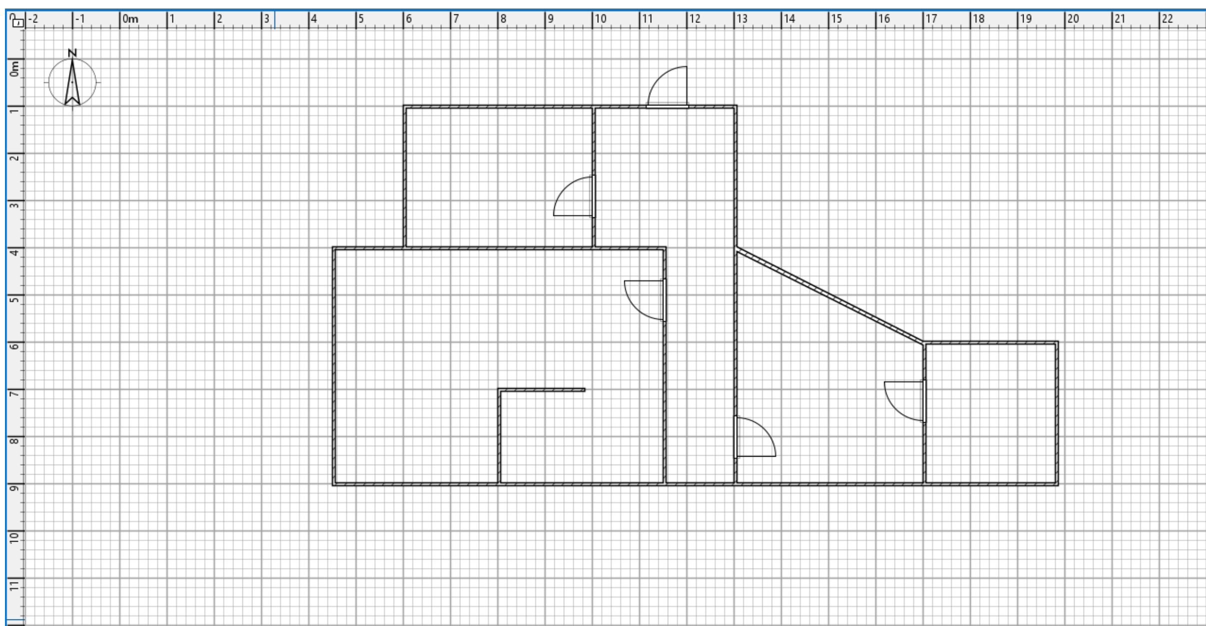
[9] https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/
[10] https://blog.unity.com/community/unityscripts-long-ride-off-into-the-sunset
[11] https://about.gitlab.com/
[12] https://git-scm.com/
[13] https://www.jetbrains.com/rider/

3.2. Hoone plaani joonestamise režiim

Esimene rakenduse vaade, millega kasutaja kokku puutub, on ruumiplaani joonestamise režiim (rakenduses „loo plaan”). Selles vaates saab ruumi seinte asukohad joonestada ja paigutada plaanile uksed ja aknad. Allpool on kirjeldatud vaate tausta implementatsiooni käik ning hoone plaani joonestamise põhiloogika.

3.2.1. Ruudustik

Paljudes joonestamis- ja CAD[14] programmides (näiteks Blender, Sweet Home 3D, Unity Probuilder) on kasutaja tegevusi abistamas mingisugune ruudustik (vt Joonis 3), mis aitab skaalast ja mõõtmetest paremini aru saada. Lisaks aitab ruudustik joondada põhilisi geomeetrilisi kujundeid (kolmnurk, ristkülik jne). Kuna suurem osa hoonetest kasutavad neid samu põhilisi kujundeid või nende kombinatsioone, lisati sama funktsionaalsus ka SOFIT Tasemeredaktor rakendusse.



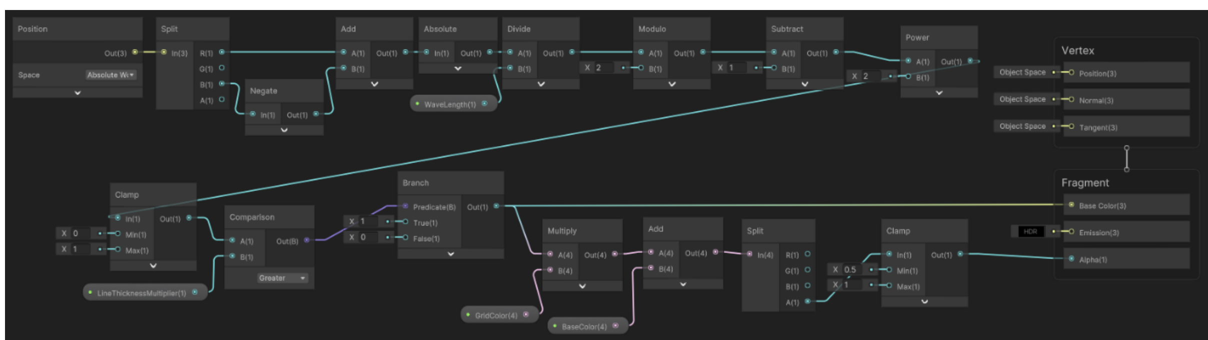
Joonis 3. Näide ruudustikust rakenduses Sweet Home 3D

Esimene iteratsioon ruudustikust oli mänguobjektide (Unity's `GameObject` komponent) põhine. Iga ruudustiku joone jaoks loodi eraldi mänguobjekt ning liigutati ruudustikus sellele vastava koha peale. Kuigi selle implementeerimine oli lihtne, oli sel lähenemisel mitu tõsist probleemi.

[14] CAD – „Computer Assisted Design” ehk raalprojekteerimine on tarkvara tüüp, mis kirjeldab rakendusi, millega saab kahe- või kolmemõõtmelisi objekte projekteerida. Sellist tüüpi rakendusi kasutatakse põhiliselt tööstuses, näiteks autode või hoonete projekteerimiseks.

Esimene neist oli jõudlus. Juba üsna tavalise kaamera suuruse juures oli vaja luua suur kogus mänguobjekte, mis aeglustas esialgset laadimisaega (sest kõik objektid tuli enne stseeni kuvamist valmis teha) kui ka käitusaja kaadrisagedust. Teiseks vajas see palju lisaarvutusi, et kuvada välja ainult need jooned, mis on parajasti ekraanil ja samas polnud see lahendus paindlik. Kui kasutaja liigutab vaadet 20 meetrit paremale, on vaja järgmise 20 meetri jooned luua. Isegi väikese kaamera pildi 2 m * 3.5 m juures on vaja luua peaaegu 80 joont (kui kasutada ruudustiku resolutsioone 5 m, 1 m, 0.5 m ja 0.1 m).

Efektiivsem ja paindlikum viis ruudustikku ekraanile joonistada on kasutada varjutajat, mis õigetel koordinaatidel olevad pikslid värvib ruudustiku joonte värvi ning ülejäänud jäävad läbipaistvaks. Unity's on lihtsasti kasutatav Shader Graph[15] liides (vt Joonis 4), millega saab visuaalprogrammeerimist kasutades kiiresti ja mugavalt üsna keerulisi varjutajaid teha. Sellise varjutaja jaoks on vaja vaid õiget matemaatilist funktsiooni. Kuna jooned peavad olema perioodiliselt, oli autori esimene mõte kasutada mõnda perioodilist funktsiooni, näiteks $\sin(x)$.

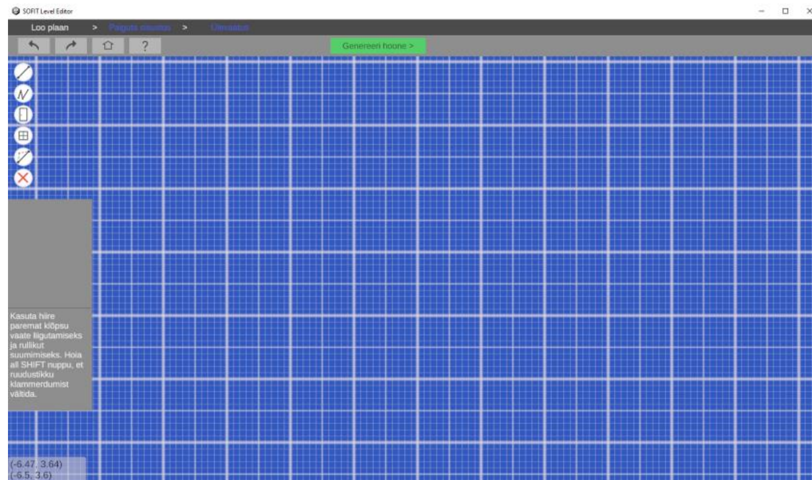


Joonis 4. Üks rakenduses kasutatav varjutaja Shader Graph tööriistas.

Esimene lahendus oligi funktsioon $y = \sin(x \cdot k)^a$, kus a on mingi kõrge aste ja k on kordaja, millega saab ruudustiku tihedust muuta. Unity dokumentatsioon aga soovitab vältida keerulisi funktsioone nagu siinus, koosinus, astendamine jne[16]. Ka seda lahendust katsetades (kõrge a väärtusega, näiteks 40) oli selgelt tunda kaadrisageduse langust.

Järgmine, tunduvalt efektiivsem varjutaja implementatsioon, kasutas mooduli funktsiooni. Joonisel 5 kuvatud ruudustiku põhifunktsiooniks on $y = ((x \cdot k \bmod 2) - 1)^2 \cdot m$, kus k ja m on muutujad, millega saab vastavalt ruudustiku tihedust ja ruudustiku joonte jämedust muuta. Erineva suurusega ruudustike kuvamine toimub vastavalt kaameras rendertatava ala suurusele. Kuna kaamera on plaani joonestamise režiimis ortograafiline, on selle suurust võimalik pärida.

[15] https://unity.com/features/shader-graph
[16] https://docs.unity3d.com/Manual/SL-ShaderPerformance.html



Joonis 5. Näide varjutajal põhinevast ruudustikust rakenduses SOFIT Tasemeredaktor.

Lisaks on mitmetel joonestusprogrammidel (näiteks Sweet Home 3D, Unity Probuilder) kasutaja abistamiseks featuur, kus näiteks joont joonistades klammerduvad joone algus ja lõpp ruudustiku ristumispunktidesse. See aitab kasutajal luua jooni, mis on ümardatud oodatava pikkuseni. Sama funktsionaalsus lisati ka SOFIT Tasemeredaktor rakendusse, kus klammerdumine toimub vastavalt kuvatavate ruudustikega. Pärast ruudustiku valmimist sai hakata lisama plaani joonestamise funktsionaalsust.

3.2.2. Plaani joonestamine

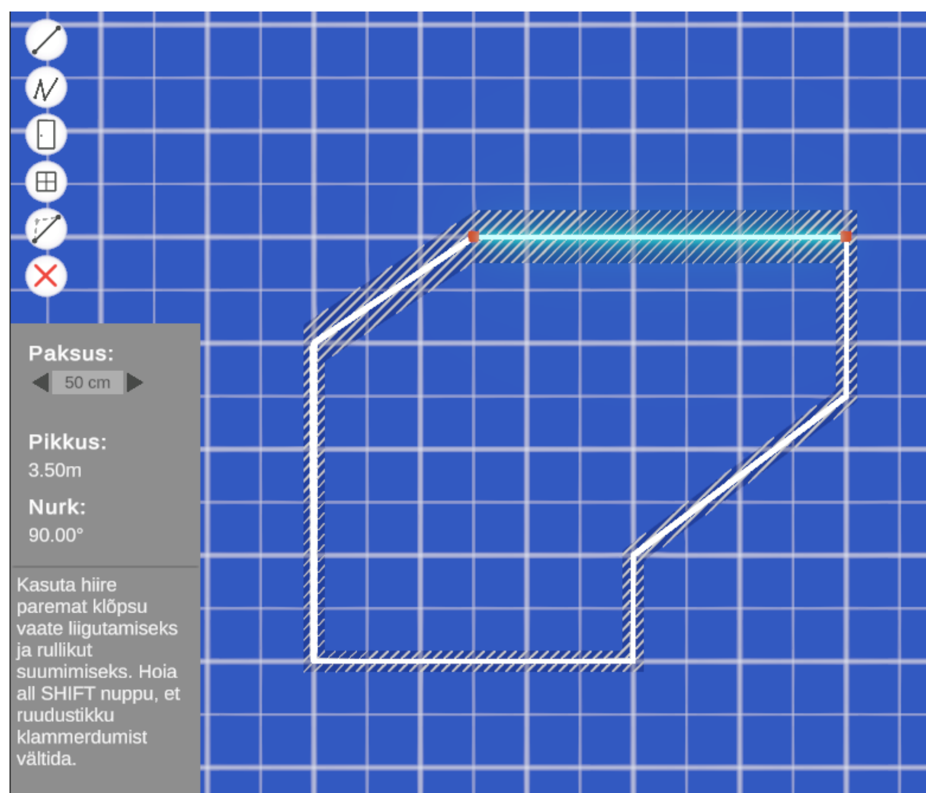
Plaani joonestamise režiimis on rakendusel seitse olekut:

1. Vaikeolek, mille käigus saab objektide peal klikkides neid valida (tõstunuppu kasutades ka mitmeid valida) ning valitud objektide omadusi (näiteks ukse laiust või seina paksust) muuta.
2. Ühe joone joonistamise olek, kus esimese kasutaja kliki peale määratakse ära joone alguspunkt ning teise kliki peale joone lõpp-punkt.
3. Mitme joone joonistamise olek, kus esimene klikk paneb paika esimese joone alguse ning iga järgmine klikk lisab jadasse uue joone. See mood on mugav kiiresti hoone piirjoonte joonestamiseks.
4. Ukse joonestamise olek, kus kursori lähedalt otsitakse lähim joon ning sobiva koha olemasolul projekteeritakse ust illustreeriv sprait vastavasse kohta joonel.
5. Akna joonestamise olek, mis toimub analoogselt ukse joonestamise olekuga, kuid sprait on erinev.

6. Joone poolitamise olek, kus saab olemasolevale joonele uue nurga lisada. Uus nurga märgis projekteeritakse sarnaselt ustele ja akendele joonele ning pärast klikki muutub olek muutmisolekuks, kus muudetavaks objektiks on just lisatud tipp.
7. Muutmise olek, kus saab joon(t)e tippu või ukse/akna asukohta muuta.

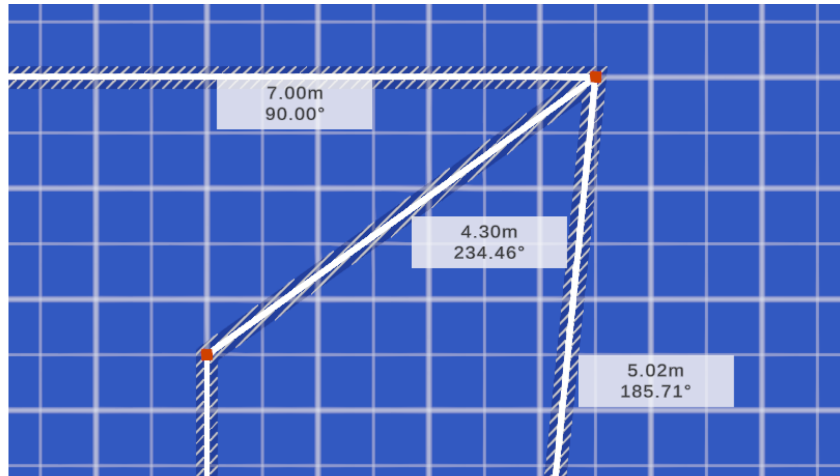
Igast olekust saab tühistamise nupuga („Esc”) või valitud oleku nuppu uuesti klikkides liikuda vaikeolekusse, sama juhtub ka pärast muutmise lõppu. Joonisel 6 on rakenduses aktiivne vaikeolek, kus on aktiivseks valitud üks seina tähistav joon (kuvatud helesiniselt) ning vasakul paneelil on näha selle seina paksus, pikkus ning suund.

Kõik rakenduses joonestatavate elementide (joon, uks ja aken) asukohad on defineeritud läbi ankurpunktide (joonisel 6 kuvatud punaste kastikestena), mida võib kutsuda ka käepidemeteks (ingl „handle”). Nende abil saab juba joonistatud joonte otspunktide ning paigutatud uste ja akende asukohti muuta.



Joonis 6. Rakenduses on vaikeolekus selekteeritud üks joon.

Joonte paigutamisel tehakse ka kontrollid – kui uue joone ots on mingi teise joone peal, siis poolitatakse see teine joon ja lisatakse sinna uus käepide (seotakse seinad omavahel). Kui käepide satub teise käepideme peale, kombineeritakse nad üheks. Uste ja akende paigutamisel kontrollitakse, et nad üksteisega ei kattuks. Kõik need kontrollid tehakse, et hilisemas võrestiku genereerimise sammus töötaks kõik ootuspäraselt.

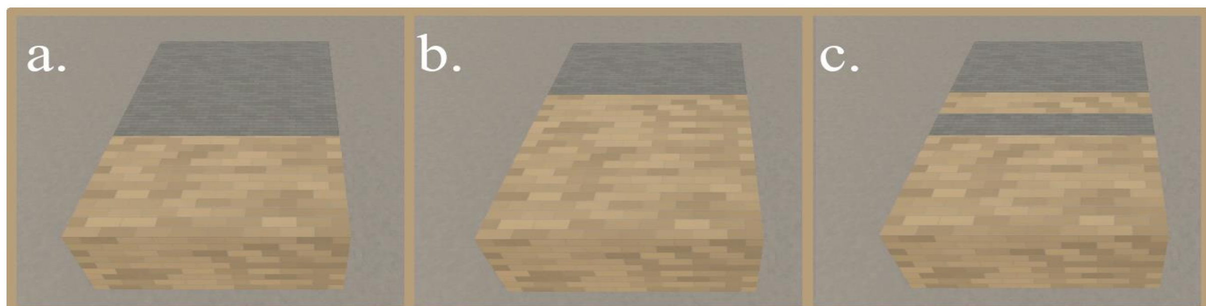


Joonis 7. Abistavad hüpikaknad joonte mõõtmete kohta.

Viimaseks abivahendiks plaani joonestamisel on abistavad hüpikvihjed joonte kohta. Nii saab kasutaja kiirelt info joone pikkuse ja nurga kohta, mitte ei pea hakkama koordinaate peast arvutama. Sellised hüpikvihjed on näha joonisel 7. Järgnevalt on kirjeldatud, kuidas kasutaja poolt märgitud joontest ning uste ja akende spraitidest genereeritakse nendele vastav 3D mudel.

3.3. Kolmemõõtmelise võrestiku loomine

Kõige lihtsam lahendus seinte võrestiku loomiseks oleks iga kasutaja poolt joonestatud joone kohta luua üks risttahukas ja sellele planeeritud avaused (uksed ja aknad) sisse „lõigata". Sellisel lähenemisel on aga mitmeid probleeme. Üks neist on võrestiku ülekattumine, mis tekib olukorras, kus kahe võrestiku ruumalad kattuvad. See on probleemiks kokkupõrgete arvutamisel ja võib tekkida ka olukord, kus kahe erineva võrestiku kolmnurgad on samal tasandil. Sellisel juhul võib keset ühte seina olla vale seina tekstuur või halvemal juhul võivad kaks tekstuuri konkureerida kuvamise nimel ja lõppkasutajale näeb see välja vilkuva tekstuurina. Töö autor on sellest toonud näite joonisel 8, kus kaks kuubikut on paigutatud osalise ülekattega. Kolmel pildil on näha, kuidas väike erinevus vaatenurgas võib kuvada kas esimese kuubiku materjali (a), teise kuubiku materjali (b) või isegi mõlema kuubiku materjale korraga (c).



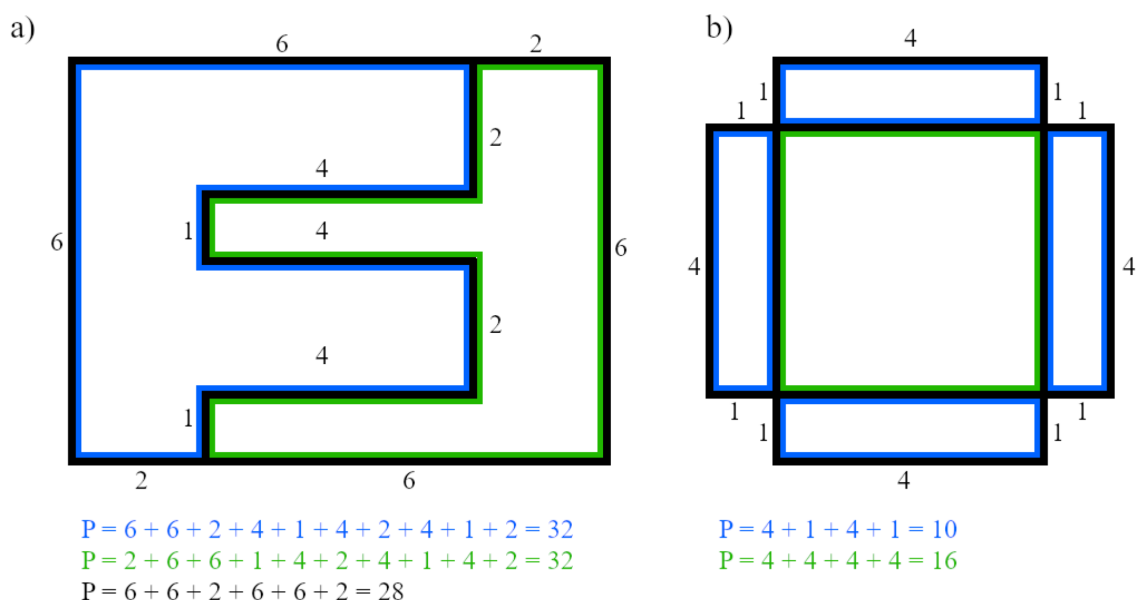
Joonis 8. Vilkuvad tekstuurid kattuvate võrestike puhul.

Lisaks on sellise lähenemisega keeruline ruumide sise- ja välispindadele erinevate materjalide määramine ning isegi kui see probleem lahendada, on siiski küsimus, kuidas teha kindlaks, milline osa risttahukast märgib välisseina ning milline siseseina. Seega, et lõpptulemus oleks ilma vilkuvate tekstuurideta ja töökindlam, on vaja natuke rafineeritumat lähenemist.

Läbi katsetuste sai valitud protsess, kus rakendus arvutab iga ruumi siseseina ja ka iga välisseina jaoks eraldi võrestiku. Nii saab välisseinale ja siseseintele (ka iga ruumi jaoks eraldi) erinevaid materjale määrata ning väikse lisaarendusega saaks ka ühe ruumi erinevaid sektsioone erineva materjaliga kuvada. Allpool on kolmes osas tutvustatud ettevalmistusi võrestamiseks, seinte võrestamist ning lõpuks põrandate ja lagede võrestamist.

3.3.1. Ettevalmistused võrestamiseks

Hoone plaani saab käsitleda kui suunamata kaalutud graafi, kus hoone seinte kohtumispunktid on graafi tippudeks ning seinad ise on graafi servadeks. Sellisest plaanist ruumide leidmine tundus esialgu töö autorile triviaalne – kui igast punktist leida seda punkti läbiv minimaalne tsükkel ja korduvad tsüklid minema visata, siis ongi leitud kõik eraldi ruumid. Pärast sellise lahenduse implementeerimist selgus aga, et on olukordi, kus see annab soovimatuid tulemusi (vt Joonis 9). Joonisel on vasakul (a) näha hoone plaan, kust selline algoritm leiab vähemalt 2 ruumi, kusjuures üks neist on alati kahte päris ruumi ümbritsev tsükkel. Paremal (b) on näha hoone plaan, kust algoritm leiaks üles neli ruumi, kuigi neid on seal viis. Seda seetõttu, et ükskõik millisest rohelisega tähistatud ruumi nurgast graafi läbimist alustades on alati mingi ruum, mille ümbermõõt oleks väiksem.



Joonis 9. Kaks näidet olukordadest, kus vähimate tsüklite leidmine ei leia kõiki ruume.

Üks lahendus võiks olla selline, mis kasutab ruumide ümbermõõdu asemel ruumide pindala, kuid keerulisemate kujundite puhul võib see olla arvutuslikult raske ja veaohtlik. Kolmas lahendus, millest sai ka töö esitamise hetkeks lõpplahendus, on järgmine:

1. Leitakse graafist (hoone plaanist) rekursiivselt kõik tsüklid ja salvestatakse andmeobjekti Room (eesti k. ruum).
2. Ruumid järjestatakse efektiivsuse tõstmiseks ümbermõõdu alusel mittekasvavalt.
3. Tsükliga vaadatakse läbi kõik leitud ruumid ning kui ruum on minimaalne, lisatakse see leitud ruumide nimekirja.

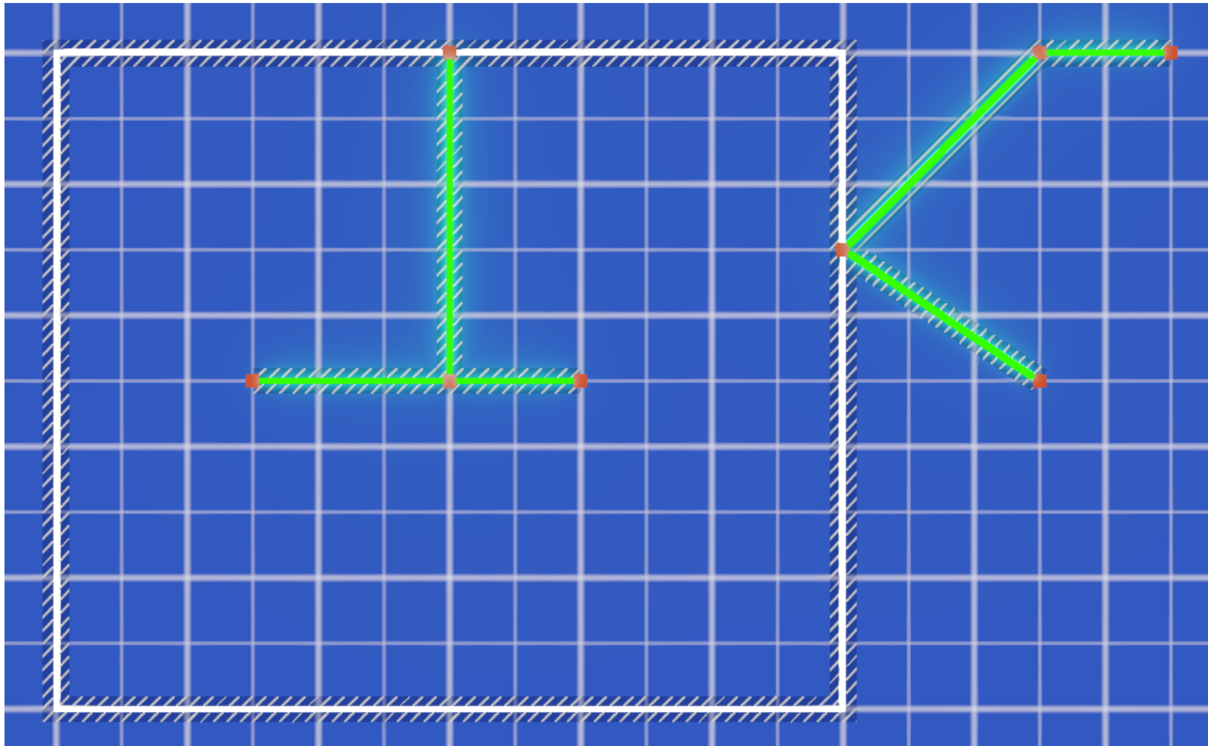
Ruumi *k* minimaalsus leitakse järgnevalt:

1. Leitakse kõik teised ruumid, millel on vähemalt üks ühine sein kontrollitava ruumiga *k* ja millel on vähemalt üks sein ruumis *k*.
2. Iga leitud ruumi *l* kohta tehakse järgnev kontroll:
   a. Kontrollitava ruumi *k* kõik seinad lisatakse nimekirja *S*.
   b. Iga leitud ruumi *l* seina *s* puhul
      i. kui sein *s* on nimekirjas *S*, eemaldatakse see nimekirjast *S*,
      ii. kui sein *s* ei ole nimekirjas *S*, lisatakse see nimekirja *S*.
   c. Kui kontrolli lõpus on nimekiri *S* tühi, on igale kontrollitava ruumi seinale leitud paariline ja järelikult on leitud mingi hulk ruume, millest saab koostada kontrollitava ruumi *k*.
   d. Kui kontrolli lõpus on nimekirjas *S* veel seinu, siis võetakse sealt üks sein, mis pole kontrollitava ruumi sein (vältimaks olukorda, kus algoritm hakkab lisama seinu, mis ei saa olla ruumi sees) ja korratakse protseduuri nii kaua kuni leitakse paariline igale seinale või kui enam pole ruume, mille seast valida.
3. Igal hetkel, kui algoritm leiab mingi kombinatsiooni ruumidest, millest saab kontrollitava ruumi *k* koostada, lõpetab algoritm töö.

Pärast kõikide ruumide leidmist on vaja leida veel „rippuvad" seinad ehk seinad, mis ei ole ühegi tsükli osa (Joonisel 10 tähistatud rohelisega). Selliste seinte leidmine on lihtne, kuna need seinad on sellised, mis pole ühegi ruumi osad. Keerulisem on aga nende grupeerimine ning seostamine kas siseruumide või välisseina(de)ga.

Kuna rippuvat tüüpi sein meenutab visuaalselt puud, oli ka ilmselge lahendus hoida neid puu andmestruktuuris, kus tippudeks on jällegi seinte kohtumispunktid ehk nurgad ja servadeks on seinad ise. Puude koostamine toimub rekursiivselt, alustades juurest ehk sellest seinast, millel

on üks ühine nurk mingi ruumi nurgaga (nurk siin tähenduses võib olla ka 180° kahe seina kohtumispunkt) ning mööda graafi edasi liikudes järgmiste seinte liitmine andmestruktuuri kuni lehttippudeni välja. Rippuvad seinad seotakse ruumidega läbi sõnastiku andmeobjekti ruumi nurkadega, kuna iga rippuv sein peab algama mingist seina nurgast (nurk taaskord võib olla ka sirgjooneline, see tähistab lihtsalt kahe või enama joone kohtumispunkti).



Joonis 10. Rippuvad seinad (märgitud rohelisega).

Välisseinte leidmine on pärast seda juba triviaalne – iga sein, mis on vaid ühe ruumi osa, on välissein. Kui veel kõikidest rippuvatest seintest eemaldada need, mis ei olnud ühegi ruumi sees, on lisaks leitud ka kõik välimised rippuvad seinad. Nüüdseks teab rakendus täpselt, kui palju ruume joonestatud hoone(te)s on ning mis seinad kuuluvad millise ruumi koosseisu. Selle infoga saab hakata seinte võrestikku genereerima.

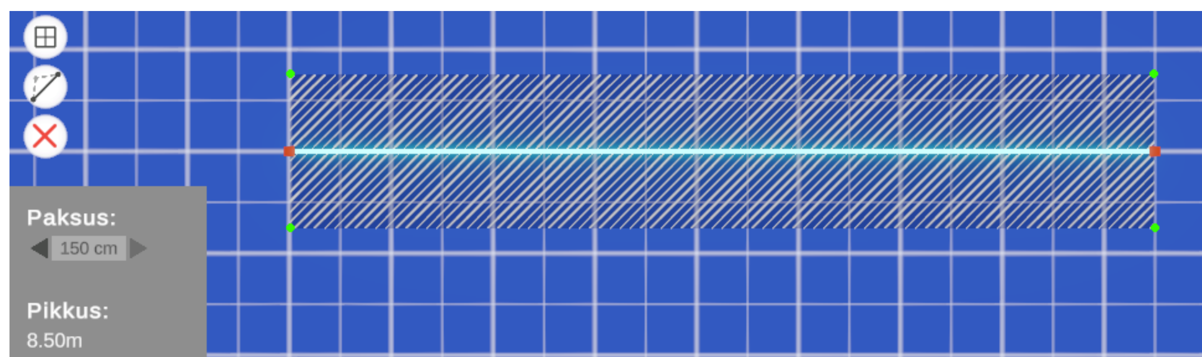
3.3.2. Seinte võrestike genereerimine

Hoone võrestiku genereerimise saab jagada kaheks suuremaks tööks: seinte genereerimine ning põranda ja lae genereerimine. Neist esimene on lihtsam, sest seinad on (vähemalt arendatud rakenduses) alati vertikaalsed ja seetõttu saab seina tahud jagada ristkülikuteks, millest igaühte saab omakorda jagada kaheks kolmnurgaks. Võrestus koosnebki punktidest või tippudest (ingl „vertex”) ja neid ühendavatest kolmnurkadest (kolmnurgad on kolmekaupa viited punktidele). Sealjuures tuleb silmas pidada, et kolmnurga tipud peavad olema kirjeldatud vaatamise suunast

päripäeva, sest võrestikud Unity mängumootoris on praagitud ehk kuvatakse vaid ühest suunast.

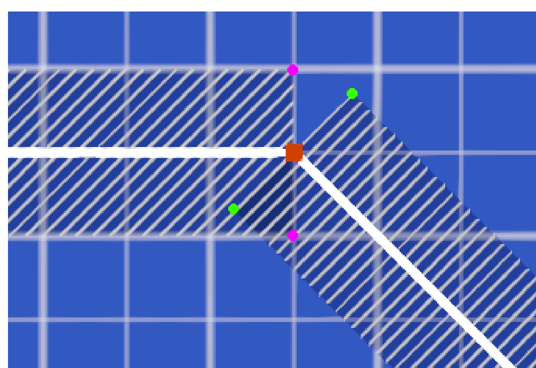
Esimene probleem seinte võrestamise puhul on see, et joon on ühemõõtmeline geomeetriline objekt, aga sein on kolmemõõtmeline, millel on lisaks pikkusele ka laius (või paksus) ja kõrgus. Seega ei saa seina võrestikku luues võrestiku punkte panna joone tippudesse, vaid neist tekkivad kolmnurgad peavad olema mingil kaugusel joonest, et tekkiv võrestik oleks kolmemõõtmeline.

Vaatleme esialgu seina kahte mõõdet (pikkus ja laius). Eraldiseisva seina puhul on võrestiku punktide määramine lihtne – need peavad olema joone tippudest joone suunaga risti mingil kindlal kaugusel joone tipust. SOFIT Tasemeredaktori puhul märgib plaanile joonistatud joon seina keskosa, seega see kaugus peab olema pool seina paksusest. Joonisel 11 oleva seina puhul on võrestiku tipud pealtvaates kujutatud roheliste täppidega.



Joonis 11. Seina kujutis plaanil. 3D võrestiku tipud on märgitud rohelisega.

Olukorras, kus kahe seina vaheline nurk ei ole sirgnurk, sellist lähenemist enam kasutada ei saa. Ühel pool seina nurka oleva terav- või nürinurga puhul satuks kummagi võrestiku punktid teineteise sisse ning teisel pool seina nurka oleva ülinürinurga puhul tekiks oodatava seina nurga asemel hoopis kitsas lõhe (vt Joonis 12).



Joonis 12. Kahe seina võrestiku tippude valesti arvutamise tulemus.

Sellisel juhul peavad võrestiku tipud olema kummagi seina piirjoonte ristumispunktis. Seda saaks lahendada trigonomeetria abil, teades kahe seina vahelist nurka ning seinte paksuseid, kuid tunduvalt lihtsam ja optimeeritum[17] lahendus on leida kummagi seina jaoks seina füüsilisi

[17] Trigonomeetrilised funktsioonid on arvutuslikult raskemad kui lihtfunktsioonid.

piirjooni kirjeldavad sirged ning leida nende sirgete lõikumispunktid. Veebilehelt cuemath.com[18] leidis autor kahe sirge ristumispunkti leidmiseks valemi (1). Valemis tähistavad x_0 ja y_0 kahe sirge ristumispunkti koordinaate ning a, b ja c on kummagi sirge joone valemite koefitsiendid ja konstant (indeks tähistab sirget).

$$(x_0, y_0) = \left(\frac{b_1c_2 - b_2c_1}{a_1b_2 - a_2b_1}, \frac{c_1a_2 - c_2a_1}{a_1b_2 - a_2b_1}\right) \quad (1)$$

$$a = y_A - y_B; b = x_B - x_A; c = x_Ay_B - x_By_A \quad (2)$$

Kasutades lisaks veebilehelt math.stackexchange.com[19] leitud valemeid (2) nende koefitsientide ja konstandi leidmiseks kahe punkti $A(x_A; y_A)$ ja $B(x_B; y_B)$ abil, oligi võimalik luua meetod kahe sirge ristumispunkti leidmiseks (vt Koodiplokk 1). See meetod kasutab tulemuse saamiseks ainult kümmet lahutustehet, 12 korrutustehet ning kahte jagamistehet. Lisaks on meetodis kontroll joonte paralleelsuse kohta, et ei tekiks nulliga jagamist.

```
public static Vector3
GetLineLineIntersectionByTwoPointsOnEitherLine((Vector3, Vector3)
firstLine, (Vector3, Vector3) secondLine)
{
    if (IsLinesParallel(firstLine, secondLine))
    {
        return GetCommonPoint(firstLine, secondLine);
    }

    float a1 = firstLine.Item1.z - firstLine.Item2.z;
    float a2 = secondLine.Item1.z - secondLine.Item2.z;

    float b1 = firstLine.Item2.x - firstLine.Item1.x;
    float b2 = secondLine.Item2.x - secondLine.Item1.x;

    float c1 = firstLine.Item1.x * firstLine.Item2.z - firstLine.Item2.x
* firstLine.Item1.z;
    float c2 = secondLine.Item1.x * secondLine.Item2.z -
secondLine.Item2.x * secondLine.Item1.z;

    float x0 = (b1 * c2 - b2 * c1) / (a1 * b2 - a2 * b1);
    float y0 = (c1 * a2 - c2 * a1) / (a1 * b2 - a2 * b1);

    return new Vector3(x0, firstLine.Item1.y, y0);
}
```

Koodiplokk 1. Meetod kahe sirge ristumispunkti leidmiseks.

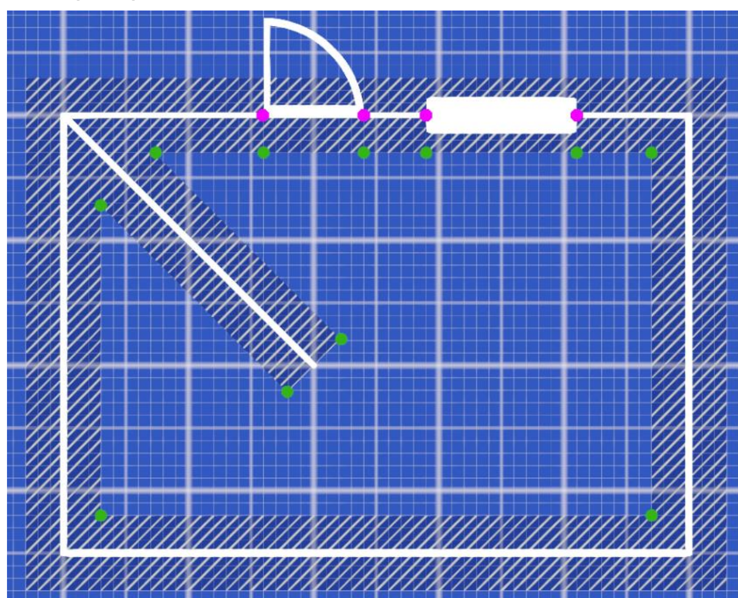
Sellega on kõik vajalik olemas, et leida iga eelnevalt leitud ruumi jaoks kahemõõtmelise seina võrestiku tipud. Nende leidmiseks tehakse läbi järgmised tegevused:

1. Luuakse nimekiri *P* tüübiga `Vector3`, milles saab salvestada kolmemõõtmelise ruumi punkte.

[18] https://www.cuemath.com/geometry/intersection-of-two-lines
[19] https://math.stackexchange.com/questions/637922/how-can-i-find-coefficients-a-b-c-given-two-points

2. Tsükliga vaadatakse läbi kõik ruumi seinad ja leitakse vaadeldava seina ja sellele eelneva seina kohtumispunkt (ruumide seinad on salvestatud päripäeva suunas).
    a. Kui selle kohtumispunktiga pole seotud ühtegi rippuvat seina, ongi otsitavaks punktiks see punkt ja lisatakse nimekirja *P*.
    b. Kui selle kohtumispunktiga on seotud üks või enam rippuv sein, siis leitakse ristumispunkt sarnaselt punktile a. eelmise seina ja esimese rippuva seina vahel (vastupäeva järjekorras), seejärel esimese rippuva seina ja teise rippuva seina vahel jne. See tegevus toimub rekursiivselt sügavuti, ehk esimese rippuva seina punktid arvutatakse enne välja kui liigutakse järgmise teise rippuva seina juurde. Lõpetuseks leitakse ja lisatakse nimekirja *P* punkt viimase rippuva seina ja vaadeldava seina vahel.
    c. Seejärel liigutakse tsükliga üle kõikide seina küljes olevate ukse ja akende objektide (järjestatuna vaadeldava seina algusest) ning lisatakse nimekirja *P* iga ukse/akna algus- ja lõpp-punktid seina peal (risti joonega, poole seina paksuse kaugusel joonest). Iga ukse ja akna juures salvestatakse lipp ka selle kohta, et sellest punktist algas aken või uks. Lisaks salvestatakse sõnastikku iga sellise punkti kohta selle punkti projektsioon joonel (vajadus selgitatud allpool).

Nende tegevuste lõpuks on iga ruumi kohta nimekiri päripäeva punktidest, millega on võimalik ruumi piirjooned joonestada. Selle tulemuse näide on kuvatud joonisel 13, kus leitud punktid on tähistatud roheliste ringidega ning uksi ja aknaid tähistavate punktide projektsioonid joonel on tähistatud lillade ringidega.



Joonis 13. Algoritmiliselt leitud ruumi siseseinte kahemõõtmelise võrestiku punktid.

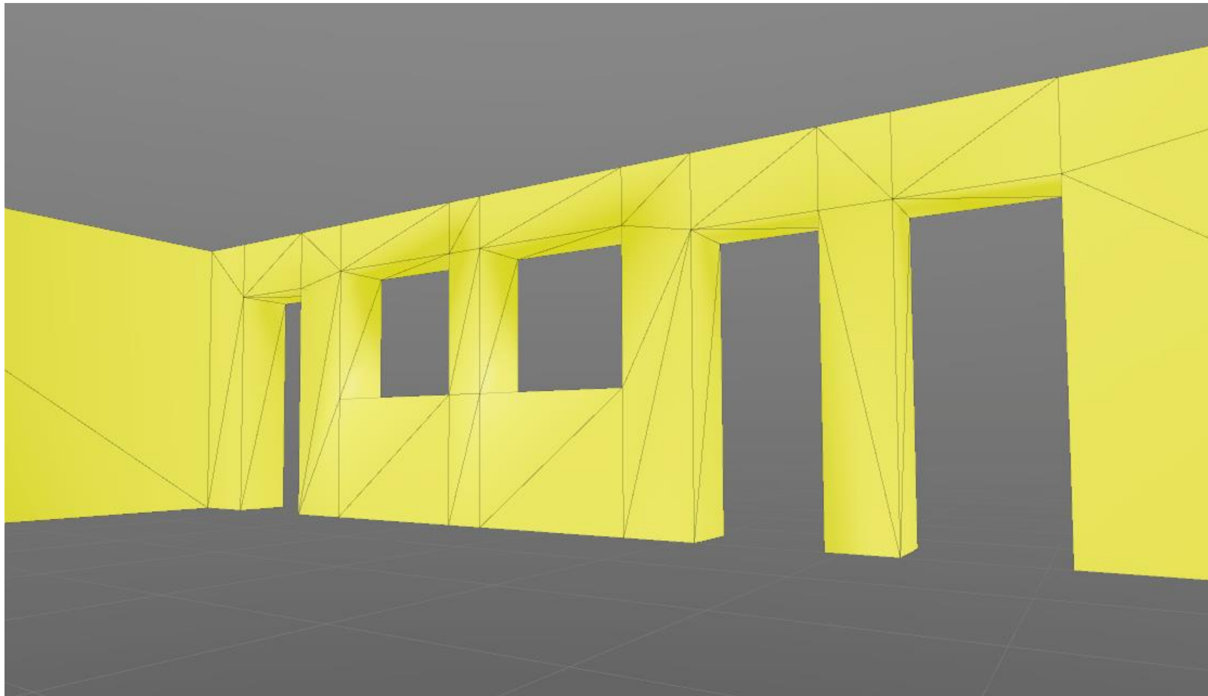
Arvutatud punktide põhjal saabki hakata võrestiku punkte ja kolmnurki arvutama. Iga punkti kohta arvutatakse eelnevalt salvestatud lippude põhjal välja selle tüüp (sein, ukse algus, ukse lõpp, akna algus või akna lõpp) ja olemasolul leitakse ka punktile vastav projektsioon joonel. Neid projektsioone on vaja selleks, et uste ja akende puhul arvutada ka võrestiku osa, mis tekitab süvendi seina. Iga sektsiooni võrestiku punktide ja kolmnurkade leidmise jaoks on loodud staatiline abiklass MeshPartGenerator. Selles klassis on üks põhiline meetod GetWallSectionMeshData(), mis arvutab ühe seina sektsiooni (joonisel 13 kahe kõrvuti oleva rohelise ringi vaheline ala on üks sektsioon) kummagi ääre tüübi põhjal välja, millist võrestikku on vaja tagastada. Seejärel kutsutakse välja üks 11-st abimeetodist, mis vastava sektsiooni tüübi tipud ja kolmnurgad tagastab. Ühe sellise meetodi näide on toodud allpool (vt Koodiplokk 2).

```
private static (List<Vector3> vertices, List<int> triangles)
GetWindowToWallSectionMeshData(Vector3 lowerVertex,
    Vector3 upperVertex, int previousSectionLowerVertexIndex)
{
    List<Vector3> vertices = new List<Vector3>();
    List<int> triangles = new List<int>();

    vertices.Add(lowerVertex);
    vertices.Add(upperVertex);

    triangles.Add(previousSectionLowerVertexIndex);
    triangles.Add(previousSectionLowerVertexIndex - 2);
    triangles.Add(previousSectionLowerVertexIndex + 2);

    triangles.Add(previousSectionLowerVertexIndex - 2);
    triangles.Add(previousSectionLowerVertexIndex - 1);
    triangles.Add(previousSectionLowerVertexIndex + 2);

    triangles.Add(previousSectionLowerVertexIndex + 2);
    triangles.Add(previousSectionLowerVertexIndex - 1);
    triangles.Add(previousSectionLowerVertexIndex + 3);

    triangles.Add(previousSectionLowerVertexIndex - 1);
    triangles.Add(previousSectionLowerVertexIndex + 1);
    triangles.Add(previousSectionLowerVertexIndex + 3);

    triangles.Add(previousSectionLowerVertexIndex - 1);
    triangles.Add(previousSectionLowerVertexIndex - 2);
    triangles.Add(previousSectionLowerVertexIndex - 4);

    triangles.Add(previousSectionLowerVertexIndex - 1);
    triangles.Add(previousSectionLowerVertexIndex - 4);
    triangles.Add(previousSectionLowerVertexIndex - 3);

    return (vertices, triangles);
}
```

Koodiplokk 2. Üks abimeetod, mis leiab sektsiooni võrestiku tipud ja kolmnurgad.

Erinevatesse sellistesse abimeetoditesse on parameetritena sisse antud alati seina alumise ja ülemise ääre tipud ja eelmise sektsiooni eelviimase tipu indeks. Seina alumist ja ülemist tippu

on lihtne leida – kuna seinad on vertikaalsed, on vaja vaid eelnevalt arvutatud punkti *y*-koordinaat ära muuta. Rakenduses on hetkel alumise koordinaadi *y*-kõrguseks 0 ja ülemise koordinaadi väärtus võetakse rakenduse sätete failist. Eelmise sektsiooni indeksit on vaja selleks, et tagastatud kolmnurkade väärtused vastaksid õigetele indeksitele. Lisaks antakse osadesse abimeetoditesse kaasa ka vajadusel ukse ülemise ääre ja/või akna alumise ja ülemise äärte kõrgused.



Joonis 14. Siseseina võrestik.

Joonisel 14 on näha eelnevalt kirjeldatud sammude tagajärjel arvutatud ruumi siseseina võrestik. Visuaalse selguse mõttes on töö autor lisanud võrestikule ühetoonilise materjali. Ruumi välisseina arvutamine toimub analoogselt, kuid ruumi põranda võrestiku arvutamist sama meetodiga lahendada ei saa.

3.3.3. Põrandate ja lagede võrestike genereerimine

Eelnevalt kasutatud meetod põrandate ja lagede (lagi on sama kujuga kui põrand, seega edaspidi räägitakse ainult põranda võrestikust) puhul ei sobi, kuna põrand ei pruugi mitte ainult ristkülikutest koosneda, vaid ei pea isegi kumer hulknurk olema. Iga ruumi põranda piirjooned on lihtne saada - need on välja arvutatud juba seinte piirjoontega (kirjeldatud eelmises peatükis), kuid kuidas neist punktidest võrestik luua?

Frey ja George [1] toovad välja viis võrestamismeetodi tüüpi:

1. Manuaalne või poolautomaatne, kus võrestiku jupid on kas ette antud või lihtsate geomeetriliste tunnustega kirjeldatavad.
2. Parametriseeritud, kus kindla kujuga punkte täis tasand vastendatakse mingi funktsiooni põhjal soovitava kujundiga.
3. Domeeni dekompositsioon, kus suure resolutsiooniga võrestikku hakatakse järjest väiksemateks osadeks jagama, jõudes lõpuks nii soovitud võrestikuni.
4. Punkti sisestamise või elementide loomise meetod, kus alustatakse soovitud kujundi piirjoontega või soovitud kujundit ümbritseva lihtsa kujundiga ning punkte või elemente lisades saavutatakse lõpuks kindlaid reegleid jälgides soovitud tulemus.
5. Konstruktiivne meetod, kus keeruline lõppvõrestik saadakse eelmist nelja meetodit kasutades arvutatud võrestikke kokku transformeerides.

Manuaalse meetodi abil valmiski seinte võrestik, kuid nagu peatüki alguses juba välja toodi, siis see meetod põranda võrestiku loomiseks ei sobi. Ka parametriseeritud meetod ei sobi, kuna sellist funktsiooni, mis iga põranda kujuga sobiks, on keeruline välja mõelda.

Domeeni dekompositsioon ja punkti sisestamise meetodid võiksid mõlemad sobivad olla ja Frey ja George kirjeldavad täpsemalt kolme levinud meetodit, mis võiksid olla sobivad kandidaadid. Esimene neist on domeeni dekompositsiooni tüüpi nelinurkpuu (ingl „quadtree“) meetod, kus soovitud kujund piiritletakse nelinurgaga ning seejärel jagatakse rekursiivselt iga nelinurk, milles on kujundi piir, neljaks kuni saavutatakse soovitav täpsus [1].

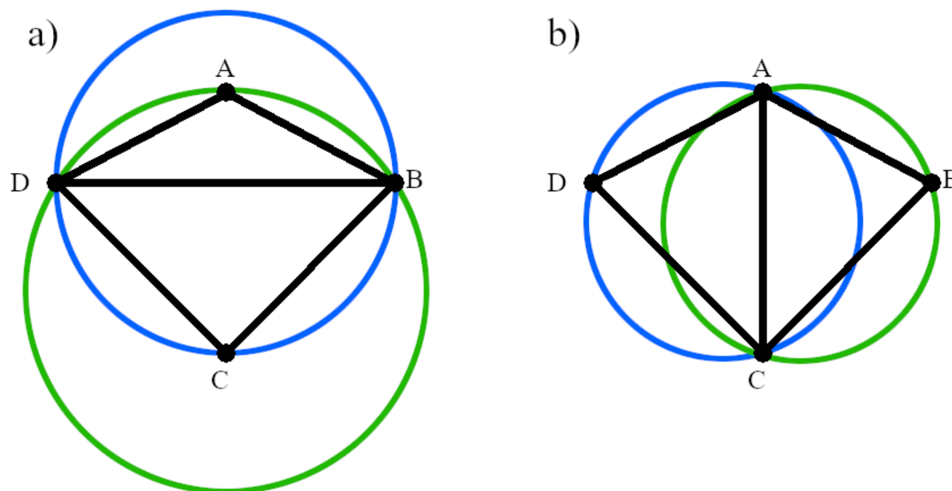
Selle meetodi eeliseks on selle lihtne implementeerimine, kuid sellega on oht, et väiksed detailid lähevad kaduma. Lisaks võib lihtsa diagonaalse joone (ruudu külgede suhtes) kirjeldamine luua suure täpsuse nõudel tuhandeid elemente, samas kui mõne muu meetodiga saab selle vaid mõne kolmnurgaga ära kirjeldada [1].

Teine meetod, mida Frey ja George täpsemalt kirjeldavad, on marssiva ääre meetod (ingl „advancing-front“), kus taaskord hakatakse kujundi ümber olevat kasti lihtsate geomeetriliste kujunditega (näiteks kolmnurgad) täitma ning kujundi piirjoonte läheduses järgitakse neid. Nii saadakse kindla resolutsiooniga tulemus. Lihtsate kujundite puhul, nagu on seda ruumi põrand, on see suhteliselt efektiivne meetod, kuid selle miinusteks on resolutsiooni valimise keerukus ja sisepunktide lisamine [1].

Resolutsioon tähendab siin kontekstis loodavate geomeetriliste kujundite (näiteks kolmnurkade) suurust. Hoone ruumi põranda detailid võivad olla mõne sentimeetri suurused,

samas võib ruum ise olla kümneid meetreid pikk ja lai. Sellisel juhul peaks kolmnurga suuruse valima üsna väikse, kuid suurem osa neist on üleliigsed, sest suur nelinurkne tühi ala ei vaja nii palju kolmnurki. Teine probleem on sisepunktide lisamine, mille arvutamine lisab keerukust ja kolmnurki võrestikku, kuid mis lihtsate suurte alade puhul midagi juurde ei anna. Neid probleeme saab küll mitigeerida, kasutades võrestiku optimeerimist[20], kuid see lisab implementeerimisele keerukust juurde.

Viimane põhiline meetod, mida Frey ja George välja toovad, on Delaunay triangulatsioon, kus võrestikku hakatakse järjest punkte juurde lisama ja iga lisatud punktiga kontrollitakse, kas Delaunay tingimus jäi püsima [1]. Delaunay tingimus on täidetud, kui iga neliku (kaks kolmnurka, mis jagavad ühte serva) puhul ühe kolmnurga punktidega kirjeldatud ringi sees ei ole neliku neljandat punkti. Kui punkti lisamine rikkus tingimuse, siis vahetatakse sisemine diagonaal teise võimaliku diagonaali vastu [1], [2].



Joonis 15. Delaunay tingimuse visualisatsioon. Vasakpoolsel joonisel ei ole tingimus täidetud ja parempoolsel on.

Delaunay tingimuse visualisatsioon on näha joonisel 15. Vasakul pool on nelik jaotatud kolmnurkadeks ABD ja DBC ning on näha, et kummagi kolmnurga tippudest joonestatud ringid sisaldavad teise kolmnurga üksikut tippu. Seega ei ole Delaunay tingimus täidetud. Paremal pool on sama nelik jaotatud kolmnurkadeks ABC ja ACD ning on näha, et tippudest joonestatud ringid ei sisalda teise kolmnurga üksikut tippu ja Delaunay tingimus on täidetud.

[20] A. Rassineux räägib sellest enda artiklis „3D mesh adaptation. Optimization of tetrahedral meshes by advancing front technique”

Delaunay triangulatsiooni eelis on see, et võrestikku ei teki tüüpiliselt kitsaid kolmnurki, sest Delaunay tingimus püüdleb võrdhaarsete kolmnurkade poole [2]. Frey ja George sõnul on Delaunay triangulatsioon kahedimensioonilise võrestiku puhul väga töökindel ja robustne ning ka üks efektiivsemaid arvutusaja suhtes. Samas toovad nad välja ka selle kõige suurema puuduse: klassikalise Delaunay triangulatsiooni puhul on lõpptulemusena saadav võrestik nürinurkne, ehk võrestikul ei saa olla sisselõikeid ja auke. Sellest puudusest mööda pääsemiseks on välja mõeldud kitsendustega Delaunay triangulatsioon (ingl „constrained Delaunay triangulation"), mis lubab võrestikku servi lisada [1].

Nende meetodite kirjelduse põhjal on selge, et parim valik on Delaunay triangulatsioon juba seetõttu, et algoritmi algandmeteks vajalikud tipud on juba eelnevalt arvutatud. Lisaks on Delaunay väga optimaalne nii kolmnurkade arvu suhtes (kolmnurki luuakse täpselt nii vähe kui vajalik) ning ka arvutuslik efektiivsus. Delaunay miinuseks on selle implementeerimise keerukus.

Frey ja George [1] toovad välja ka üldise skeemi Delaunay triangulatsiooni implementeerimiseks, kuid mitmed teised autorid[21], on avaldanud konkreetseid algoritme, mis kas lihtsustavad algoritmi, teevad seda kiiremaks või lisavad robustsust (või mõni kombinatsioon kolmest). Käesoleva töö autor valis enda implementatsiooni jaoks Sloan'i 1987. ja 1993. aastatel avaldatud kaks artiklit, millest esimeses toob ta välja üsna lihtsa algoritmi kumera Delaunay triangulatsiooni arvutamiseks ning teises artiklis algoritmi selle kumera triangulatsiooni töötlemiseks mitte-kumeraks [3], [4]. Kuigi mõlemad artiklid on üsna ammu kirjutatud, on neis olevad algoritmid ühed lihtsamad, millega käesoleva töö autor kokku puutus ja kuna loodavad võrestikud on üsna lihtsakoelised, ei olnud mõistlik keerulisemaid algoritme implementeerima hakata. Sloan'i algoritmide kasuks rääkis ka see, et internetis leidus kaks

[21] Mittetäielik nimekiri erinevatest Delaunay algoritmide variantidest:
1) Green, P. J. and Sibson, R. Computing Dirichlet tessellations in the plane, The Computer Journal, (1978)
2) Lawson, C. L. Software for C^1 interpolation, in Rice, J. (ed.) Mathematical Software III, Academic Press, New York, (1977)
3) Lee, D. T. and Schachter, B. J. Two algorithms for constructing a Dehunay triangulation, International Journal of Computer and Information Sciences, (1980)
4) Bowyer, A. Computing Diriehlet tessellations, The Computer Journal, (1981)
5) Watson, D. F. Computing the n-dimensional Delaunay triangulation with application to Voronoi polytopes, The Computer Journal, (1981)
6) Cline, A. K. and Renka, R. L. A storage efficient method for construction of a Thiessen triangulation, Rocky Mountain Journal of Mathematics, (1984)
7) Tianyun Su, Wen Wang, Zhihan Lv, Wei Wu, Xinfang Li, Rapid Delaunay triangulation for randomly distributed point cloud data using adaptive Hilbert curve, Computers & Graphics, Volume 54, 2016
8) Shewchuk J.R. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. 1996

väga head üsna hiljutist materjali nende implementatsioonide kohta. Üks neist programmeerimiskeeles C[22] ja teine C#[23].

Sloani algoritmil on enne põhiosa kaks mittekohustuslikku ettevalmistavat sammu: normaliseerimine ja punktide lahterdamine. Neist esimene skaleerib etteantud punktid lõiku $[0, 1]$ x-teljel ja $[0, 1]$ y-teljel, kusjuures kuvasuhe jääb originaalseks, ehk kui punktid on originaalselt venitatud mööda x-telge, siis on nad seda ka pärast normaliseerimist. Punktide lahterdamisel jagatakse need kastidesse ja sorteeritakse selle põhjal, et hiljem punkte lisama hakates saaks järjest lisada üksteisele lähedal olevaid punkte [3].

Käesoleva töö autor otsustas implementeerida neist kahest sammust vaid esimese, kuna see on üsna lihtne (Sloan toob selleks isegi valemid välja) ja see lihtsustab veel ühte hilisemat algoritmi sammu. Teist optimisatsiooni ei implementeeritud, kuna see toob põhilise kasu suvaliselt paigutatud punktide puhul. SOFIT Tasemeredaktor rakenduse puhul on aga punktid juba eelnevalt järjestatud, mistõttu erilist kasu see tuua ei saa.

Algoritmi põhiosa alguses soovitab Sloan luua super-kolmnurga ehk väga suure kolmnurga, mille sisse kõik punktid mahuvad. Tema soovitus on luua kolmnurk koordinaatidega $(-100;\ 100)$, $(100;\ -100)$ ja $(0;\ 100)$ [3]. Kuna kõik algoritmis kasutatavad punktid on normaliseeritud, siis saab kindel olla, et kõik punktid on selle kolmnurga sees. Järgmiseks lisab algoritm ükshaaval võrestikku punkte järgmise eeskirja järgi:

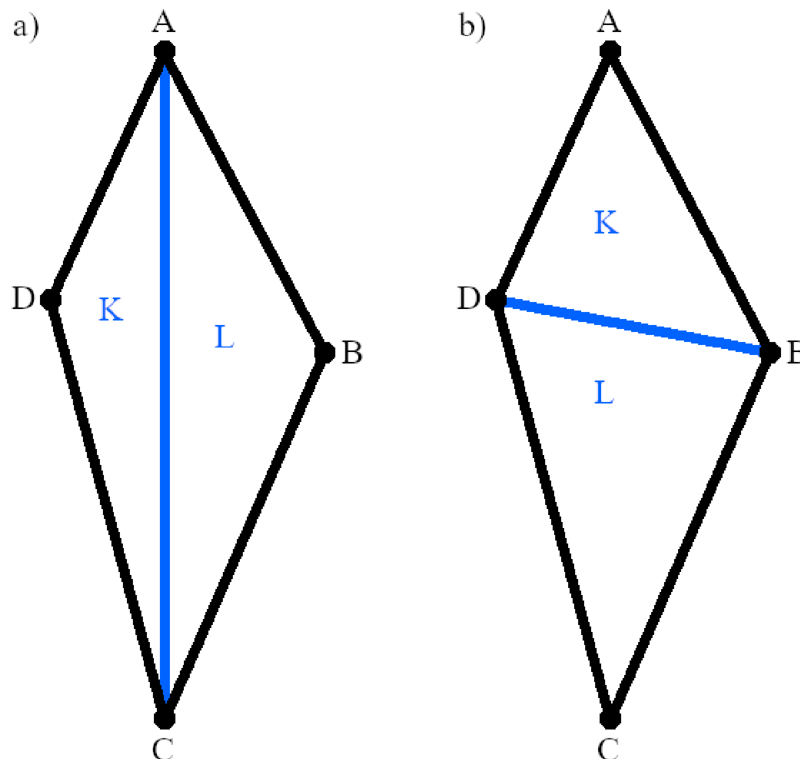
1. Leiab olemasoleva kolmnurga K, mille sees lisatav punkt on ja eemaldab selle kolmnurkade järjendist.
2. Jagab selle kolmnurga kolmeks, olgu uued kolmnurgad K_1, K_2 ja K_3, kusjuures iga kolmnurga üks tipp on just lisatud punkt.
3. Lisab kolmnurgad K_1, K_2 ja K_3 pinusse ning senikaua kuni pinus on kolmnurki teeb iga kolmnurga jaoks järgmised tegevused:
    a. Eemaldab pinu kõige pealmise kolmnurga ja kontrollib, kas Delaunay kriteerium kehtib selle iga ümbritseva kolmnurga kohta. Delaunay kriteeriumi kontroll toimub Cline ja Renka meetodist laenatud kontrolli põhjal (täpsemalt allpool).
    b. Kui Delaunay tingimus kehtib, siis jätkab tsükkel tegevust järgmise pinus oleva kolmnurgaga.

[22] https://github.com/xmdi/CAD-from-Scratch videod 16 ja 17.
[23] https://forum.unity.com/threads/programming-tools-constrained-delaunay-triangulation.1066148

c. Kui tingimus ei kehti, kasutatakse Lawson'i algoritmist laenatud neliku (kontrollitav kolmnurk ja selle naaber moodustavad neliku) diagonaali vahetust (vt Joonis 16) ning mõlemad kolmnurgad lisatakse pinusse.

4. Eemaldab kõik punktid, mis olid varasemalt loodud super-kolmnurga osa ning kõik kolmnurgad, mille vähemalt üks tipp oli mõni neist punktidest.

Terve ülal kirjeldatud algoritm on võetud Sloan'i artiklist [3].



Joonis 16. Lawsoni algoritmi näidis. Algoritm vahetab diagonaali nii, et kahest ühisest tipust (A ja C) liigutatakse ühe kolmnurga puhul üks tipp teise kolmnurga eraldiseisvasse tippu ning teise kolmnurga puhul teine tipp esimese kolmnurga eraldiseisvasse tippu.

Kogu protsessi lihtsustamiseks lõi töö autor klassi DelaunayTriangle, milles on kirjeldatud kolmnurga kolm tippu läbi indeksite, iga külje vastas oleva naaberkolmnurga indeks ja abimeetodid. Indeksid viitavad põhiklassis DelaunayTriangulation olevatele järjenditele, milles on punktid ja kolmnurgad. Nii saab sama DelaunayTriangle objekt viidata korraga nii normaliseeritud kui ka normaliseerimata punktidele ja kõik võrdlused toimuvad täisarvude, mitte objektide, vahel.

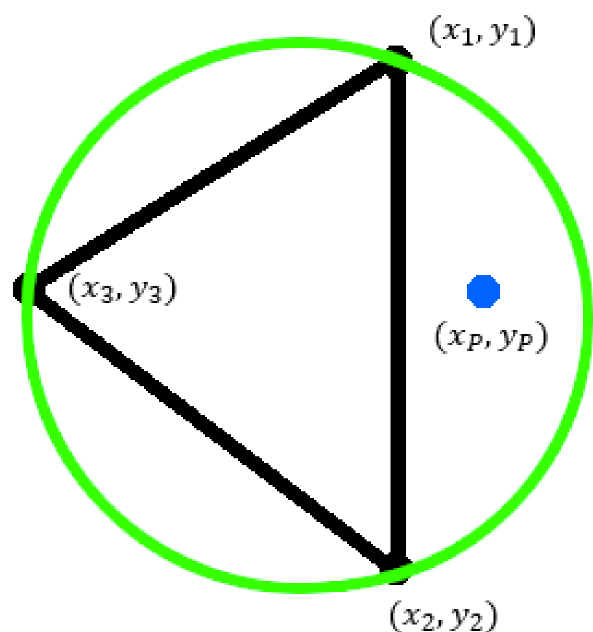
Algoritmi punktis 1 kirjeldatud kolmnurga leidmise meetod on lihtne: alustatakse viimati lisatud kolmnurgast ja kui punkt ei ole selle kolmnurga sees, siis leitakse, milline selle külgedest on suunatud otsitava punkti poole. „Selle poole suunatud" tähendab siin kontekstis, et punkt on sellest küljest vasakul, arvestades, et kolmnurga küljed on päripäeva. Seejärel

korratakse tegevust, võttes aluseks leitud naaberkolmnurga, kuni leitakse kolmnurk, mis sisaldab otsitavat punkti.

Sloan'i algoritmi kõige tähtsam osa on Delaunay tingimuse säilitamine ning selle originaalne viis oli ühe kolmnurga tippudest ring projekteerida ning kontrollida, kas neliku viimane punkt on selle ringi sees [1]. Sloan [3] kasutab aga Cline ja Renka [5] arvutuslikult lihtsamat, kuid siiski väga robustset testi:

1. Arvuta $CosA = x_{13}x_{23} + y_{13}y_{23}$ ja $CosB = x_{2P}x_{1P} + y_{2P}y_{1P}$, kus
   a. $x_{13} = x_1 - x_3;\ x_{23} = x_2 - x_3;\ y_{13} = y_1 - y_3;\ y_{23} = y_2 - y_3;$
   b. $x_{1P} = x_1 - x_P;\ x_{2P} = x_2 - x_P;\ y_{1P} = y_1 - y_P;\ y_{2P} = y_2 - y_P;$ (vt Joonis 17)
2. Kui $CosA \geq 0$ ja $CosB \geq 0$, siis neliku diagonaal vahetust ei vaja, vastasel juhul kontroll jätkub.
3. Kui $CosA < 0$ ja $CosB < 0$, siis neliku diagonaal vajab vahetust, kontrolli võib lõpetada.
4. Arvuta väärtused $SinA = x_{13}y_{23} - x_{23}y_{13}$ ja $SinB = x_{2P}y_{1P} - x_{1P}y_{2P}$ ning nende põhjal $SinAB = SinA \cdot CosB + SinB \cdot CosA$.
5. Kui $SinAB < 0$, siis vajab neliku diagonaal vahetust, vastasel juhul mitte. Kontroll on lõppenud [3], [5].

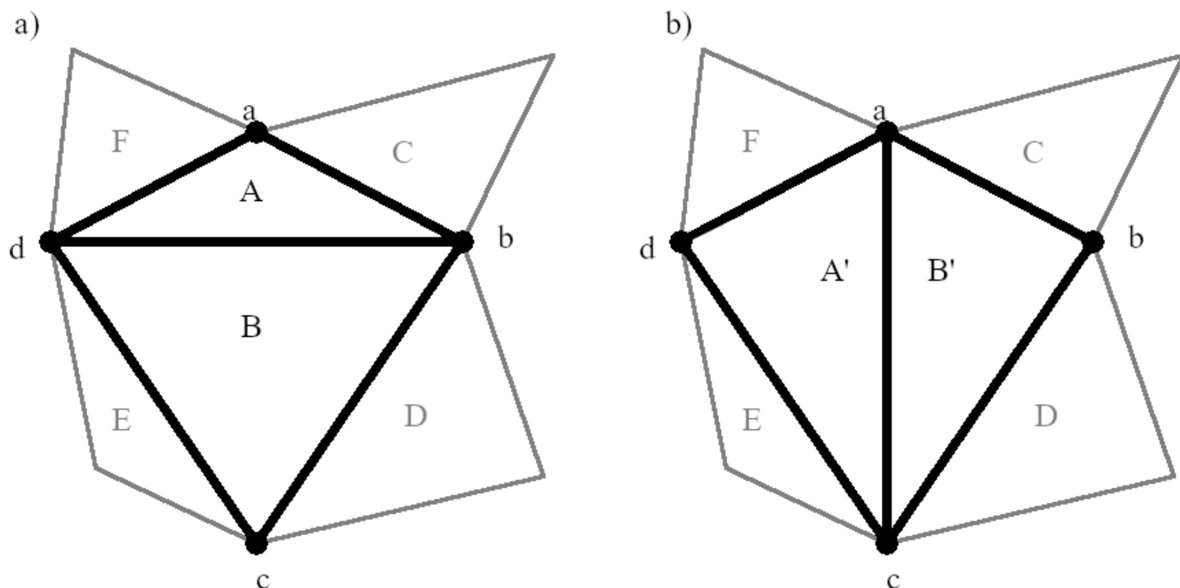
Neliku diagonaali vahetamist Lawson'i meetodil on juba ülevalpool kirjeldatud, kuid Sloan mainib ka ära, et tähtis on pärast diagonaali vahetust kontrollida ka ümbritsevate kolmnurkade viiteid [3]. Kolmnurkade viidete vahetus ei ole keeruline. On vaja jälgida, kuidas toimus vahetus ning vastavalt vahetada kummagi kolmnurga naaberkolmnurkade viited ümber. Joonis sellise vahetuse ning naaberkolmnurkade muutuste kohta on allpool (vt Joonis 18).



Joonis 17. Cline ja Renka algoritmis kasutatavad tähised. Punkt (x_P, y_P)on kontrollitav punkt.

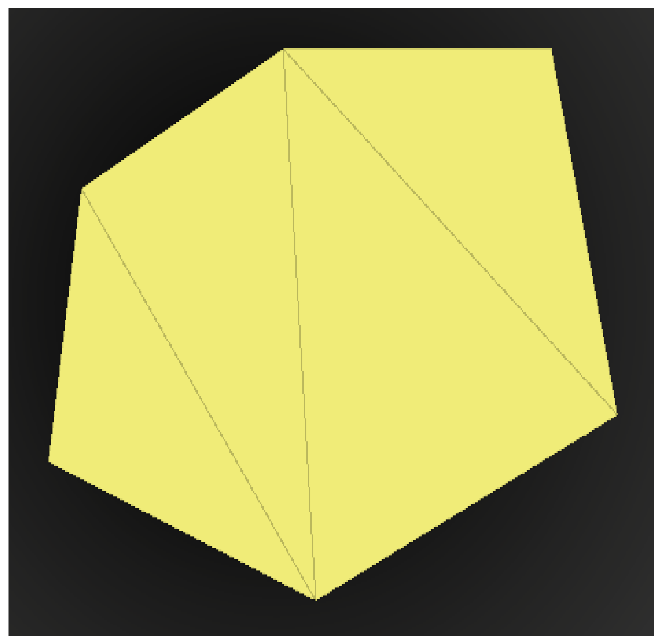
Lõpetuseks on vaja veel eemaldada algoritmi alguses loodud super-kolmnurga tipud ja nendega seotud kolmnurgad. Kuna super-kolmnurga tippude indeksid on teada (need lisati normaliseerimise lõpus punktide järjendi

lõppu), siis on nende eemaldamiseks vaja kõik kolmnurgad üle vaadata ja iga kolmnurk, mille ükski tipp ei ole esialgse super-kolmnurga tipp, on lõpliku triangulatsiooni kolmnurk. Üks selline leitud ebakorrapärase kujundi triangulatsioon on joonisel 19.



Joonis 18. Neliku diagonaali vahetusel toimuvad naabrite muutused. Vasakul (a) on kujutatud nelik enne diagonaali vahetust. Kolmnurk A naabriteks on C, B ja F. Pärast diagonaali vahetust (b) on kolmnurga A' naabriteks B', E ja F.

Kitsendustega Delaunay triangulatsiooni loomiseks on kaks lähenemist: alguses arvutada kujundi piirjooned ja seda arvesse võttes luua võrestik või luua esialgu tavaline Delaunay võrestik ning seejärel see mitte-kumeraks muuta [4]. Sloan on avaldanud efektiivse algoritmi teisele lähenemisele, mis toetub küll ta enda 1987. aastal avaldatud algoritmile, kuid tegelikkuses pole vahet, kuidas esialgne Delaunay triangulatsioon arvutati. Tähtis on veel arvesse võtta, et seda algoritmi saab kasutada vaid kahedimensiooniliste triangulatsioonide jaoks, kuid kuna eelnevalt loodud võrestik ongi kahemõõtmeline, siis see probleeme ei tekita [4].

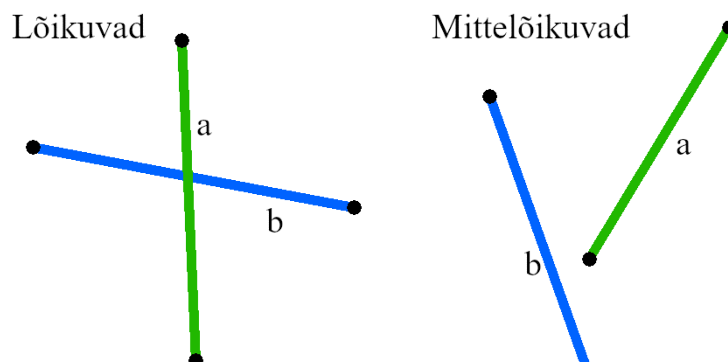


Joonis 19. Kumera ebakorrapärase kujundi Delaunay triangulatsioon.

Algoritmi sisendiks on triangulatsioon *T*, ehk tipud ja neid ühendavad kolmnurgad ning järjend kujundi servadest. Algoritm ise on järgnev:

1. Iga kujundi serva *s* puhul tehakse järgmised tegevused:
   a. Leitakse kõik triangulatsiooni servad, mis lõikuvad servaga *s* ja lisatakse järjekorda *L*. Kui serv *s* on juba triangulatsiooni *T* osa, pole samme *b* ja *c* vaja teha ja saab liikuda järgmise serva juurde.
   b. Eemaldatakse järjekorrast *L* lõikuv serv *l* ja tehakse kontroll: kui kahest kolmnurgast, mille ühine serv on *l*, koosnev nelik on
      i. kumer, siis vahetatakse selle neliku diagonaal (samamoodi nagu eelmises algoritmis) ja kui see uus diagonaal jätkuvalt lõikab serva s, lisatakse see järjekorda *L* ning kui ei lõika, lisatakse see nimekirja *U*.
      ii. mitte-kumer, siis lisatakse serv *l* järjekorra *L* lõppu ning tehakse kontroll järgmise servaga.
   c. Kui kõik servad järjekorrast *L* on töödeldud, käiakse tsükliga üle nimekirja *U* ning iga serva *u* kohta kontrollitakse, kas nelik, mille diagonaaliks on *u*, täidab Delaunay kriteeriumi. Kui ei täida, vahetatakse diagonaal u diagonaali u′ vastu (taaskord identselt eelmises algoritmis kirjeldatud vahetusega).
2. Viimase sammuna kontrollitakse iga kolmnurga puhul, kas see on piirjoontega määratud kujundi sees ning kui ei ole, eemaldatakse see triangulatsioonist [4].

Algoritmi esimene osa (1a ja 1b) veendub, et triangulatsiooni tekiks serv *s* ning teine osa (1c) optimiseerib tekkinud triangulatsiooni, et see vastaks Delaunay tingimusele. Sloan väidab, et esimeses osas tehtavad tegevused leiavad alati olukorra, kus triangulatsioonis on serv *s* [4].



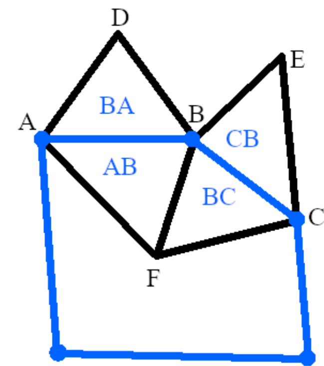
Joonis 20. Vasakul oleva kahe ristuva lõigu puhul on näha, et lõigu *a* kummastki punktist vaadeldes on üks lõigu *b* tipp paremal ja teine vasakul pool lõiku *a*. Sama kehtib ka lõiguga *b*. Paremal olevad kaks lõiku aga ei ristu ning on ka näha, et kuigi lõigu *a* puhul on endiselt lõigu *b* otspunktid kummalgi pool lõiku *a*, kuid lõigu *b* puhul on mõlemad lõigu a otspunktid ühel pool lõiku *b*.

Lisaks olemasolevatele abimeetoditele oli vaja implementeerida kaks uut: meetod kontrollimaks, kas kaks serva lõikuvad, ning meetod, mis kontrollib neliku kumerust. Mõlemad lahendused said inspiratsiooni Youtube kasutajalt xmdi[24], kes enda CAD rakenduse loomise videotes kasutas samuti Sloan'i algoritmi. Kaks serva ristuvad, kui serva a puhul on serva *b* üks tipp vasakul pool serva *a* ja teine tipp paremal pool serva *a* ning serva *b* puhul on üks serva *a* tipp paremal ja teine vasakul pool serva *b* (vt Joonis 20).

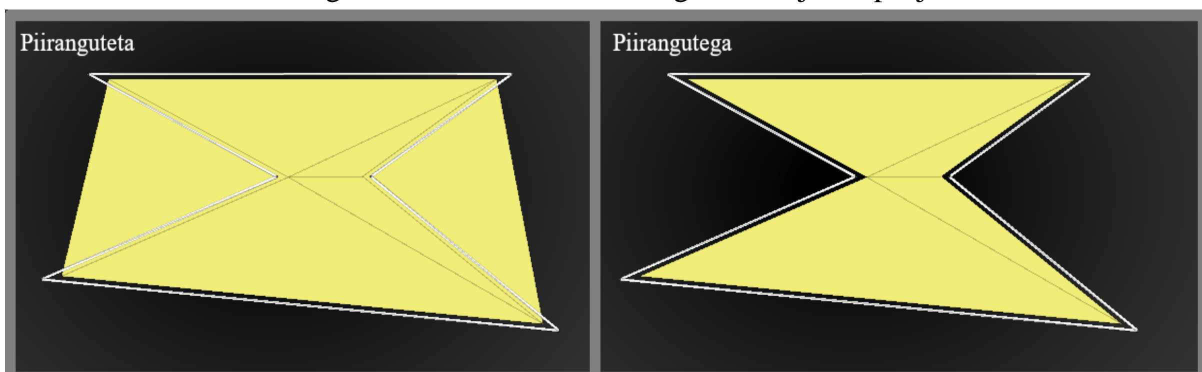
Neliku kumeruse kontrollimine on sarnane. Kui mööda neliku servi päripäeva liikudes on iga järgmine serv eelmisest samas suunas (paremal või vasakul eelmisest servast), pole ükski vaadeldava nelinurga sisenurk üle 180° ning nelinurk on kumer.

Lõpetuseks on vaja kustutada kõik kolmnurgad, mis pole piirjoontega määratud kujundi sees. See on lahendatud järgmise kahe sammuga:

1. Iga piirjoone serva jaoks leida kolmnurk, mille üks külg on see serv ning mille ülejäänud kaks serva on kujundi sees. Selleks on Unity foorumite kasutaja ThundThund välja pakkunud elegantse lahenduse: kuna kõik kolmnurgad ja piirjoon on päripäeva, siis on ilmselge, et kujundi sees on just need kolmnurgad, mille piirjoonega ühine serv on defineeritud ka päripäeva (vt Joonis 21). Lisame need kolmnurgad pinusse.
2. Kuni pinus on kolmnurki, eemaldame sealt ühe ning lisame selle lubatud kolmnurkade nimekirja. Seejärel lisame pinusse kõik naaberkolmnurgad, välja arvatud need, mille mingi serv on piirjoone mõni serv või mis on juba varem vaadeldud. See tagab, et kõik leitud kolmnurgad on kujundi piirjoonte sees.



Joonis 21. Sinise piirjoonega määratud kujundi sise-kolmnurkade leidmine. Iga kolmnurga sisse on kirjutatud tema ja kujundi piirjoone ühine serv.



Joonis 22. Kahe Sloan'i algoritmi järel tekkinud võrestikud.

[24] https://www.youtube.com/channel/UCDNCqs2pvtW3S-Iys-2ATMw

Joonisel 22 on vasakul pool näha esimese Sloan'i algoritmiga loodud kumer triangulatsioon ning paremal pool Sloan'i teise algoritmi järel tekkinud mitte-kumer algoritm. Töö autor on selguse mõttes lisanud joonistele ka oodatava kujundi piirjooned.

Peatükk 3.3 on selle lõputöö üks mahukamaid ja ka võrestike genereerimise implementatsiooni peale kulus kõige suurem osa ajast. Seda osalt seetõttu, et tegemist on üsna keeruliste algoritmidega, kuid samas saab vastulausena tuua, et ülejäänud rakenduse arendus oli mõneti lihtsam tänu kasutatud programmeerimismustritele. Järgmises alapeatükis ongi sellest pikemalt räägitud.

3.4. Kasutatud programmeerimismustrid

Raamat „Design Patterns: Elements of Reusable Object-Oriented Software" on tõenäoliselt üks enimloetud programmeerimisalaseid raamatuid. Selle autorid Erich Gamma, Richard Helm, Ralph Johnson ja John Vlissides kirjeldavad programmeerimismustreid kui edukate disainide ja arhitektuuride taaskasutamist probleemide puhul, mida on juba varem korduvalt lahendatud. Nende sõnul oskab kogenud disainer eelnevalt edukaid mustreid kasutades probleeme lahendada enne, kui need tekkinud on. Gamma jt. on enda raamatus välja toonud levinumad programmeerimismustrid ning kirjeldavad ära millal neid kasutama peaks ja mis võivad olla tagajärjed, mida tuleb kaaluda [6]. Robert Nystrom kasutab seda raamatut baasiks enda omale kus ta kirjeldab mänguarenduse spetsiifilisi probleeme ja mustreid, millega neid lahendada [7].

SOFIT Tasemeredaktor rakenduse arenduse käigus pidi töö autor lahendama mitmeid ülal mainitud raamatute poolt kirjeldatud probleeme. Selle asemel, et varem lahendatud probleeme uuesti lahendada, kasutati nende lahendamiseks juba olemasolevaid lahendusi. Lisaväärtust loob see, et rakenduse arendust jätkab Criffin ning üldkasutatavad mustrid ja konventsioonid teevad koodibaasi ülevõtmise lihtsamaks.

Esimeses alapeatükis on välja toodud olekumustri kirjeldus ja implementatsioon rakenduses, järgmises alapeatükis aga Singleton[25]'i mustri head ja vead ning vigadest hoolimata selle kasutamise põhjused. Kolmandas peatükis kirjeldatakse teenuseotsija mustrit. Eelviimases ja viimases peatükis tutvustatakse lõpetuseks põgusalt jälgija- ja käsumustrit.

[25] Autor ei leidnud ühtegi head eestikeelset tõlget sellele mustrile, seega kasutatakse töös selle ingliskeelset nimetust.

3.4.1. Olekumuster

Lõplik automaat ehk lõplik olekumasin on abstraktne matemaatiline mudel, mis koosneb lõplikust hulgast olekutest, üleminekutest ühest olekust teise, algolekust ning võimalikest sisenditest [8]. Olekumuster on lõpliku olekumasina implementatsioon programmaatiliselt [7]. Seda kasutatakse tüüpiliselt siis, kui rakenduses on mingi arv olekuid, millest ainult üks saab korraga aktiivne olla. SOFIT Tasemeredaktor rakenduse puhul on selliseid olukordi kaks: režiimide olekud (plaani joonestamine, objektide paigutamine, tulevikus ka ülevaatus ja eksport) ja plaani joonestamise režiimi olekud (joone joonistamine, ukse/akna paigutamine jne).

Iga plaani joonestamise olek on defineeritud klassina, mis pärib meetodid ActivateState() ja DeactivateState() abstraktsest ülemklassist DrawModeState. Olekuvahetusel kutsutakse kõigepealt välja vana oleku DeactivateState() meetod ning seejärel uue oleku ActivateState() meetod. Näiteks joone joonistamist alustades liigub rakendus LineDrawState olekusse, mille aktiveerimisel eemaldatakse kõik valitud objektid valikust ja luuakse uus joone alguspunkt.

3.4.2. Singleton'i muster

Singleton'i muster täidab korraga kahte ülesannet: veendub, et klassist on vaid üks instants ning tekitab globaalse viite klassi [6]. Samas hoiatab selle kohta Nystrom, et see võib mõnikord tuua rohkem kahju kui kasu. See disainimuster on tema sõnul ohtlik seetõttu, et see võib testimist raskendada (kui klassi võidakse kasutada igalt poolt projektist, on raske leida vigast kohta), see soodustab seoste tekkimist mitte seotud klasside vahel ning see ei toeta mitmelõimelisust. Nystrom soovitab singleton'i mustri asemel kasutada staatilist klassi või teenusepakkuja (ingl „service locator") mustrit [7].

Nystromi poolt pakutud klassi staatiliseks tegemist ei saa Unity's kasutada, kui on soov, et sama klass kasutaks ka mõnda MonoBehaviour ülemklassi meetodeid (näiteks Start() või Update()). Need meetodid on klassimeetodid, mistõttu staatilises klassis neid kasutada ei saa. Seetõttu kasutas töö autor mitmes kohas singleton'i mustrit, kuid üritas võimalusel kasutuskohti minimaliseerida. Näiteks mitmed kontroller- või administraatorklassid, mis tüüpiliselt võiksid olla singleton'i mustriga implementeeritud, on loodud hoopis tavaklassidena ning ligipääs neile on läbi ühe singleton tüüpi klassi (GameManager). See kindlustab, et läbi selle klassi neid kasutades on vaid üks instants saadaval. Nystromi soovitatud teenuseotsija mustrit aga töö autor ühes kohas ka kasutas. Sellest järgmises alapeatükis.

3.4.3. Teenuseotsija muster

Teenuseotsija mustri tööpõhimõte on lihtne. Nystrom'i sõnul koosneb implementatsioon kolmest osast: teenuse liidesest, teenusepakkujast ja teenuse otsijast. Rakenduse käivitusajal leiab teenuse otsija mingisuguse loogika põhjal ühe teenuse (või vastava teenuse puudumisel vaiketeenuse) ja registreerib selle teenusepakkuja juures, kusjuures teenusepakkuja ise ei tea teenuse leidmise loogikast ega isegi teenuse tüübist midagi [7].

SOFIT Tasemeredaktor rakenduses täpselt Nystromi näite järgi mustri implementatsioon tehtud ei ole, kuid põhimõte on seal sama. Rakenduses on tõlkeid pakkuvad klassid EstonianTranslation ja EnglishTranslation, mis on abstraktse klassi Translation alamklassid. Need käituvad mustri mõttes kui teenused. Rakenduse käivitumisel leiab GUIController klass vastavalt sätete failis märgitud keelekoodile õige tõlkeklassi ning edaspidised tõlgete tekstide otsimised käivad läbi GUIController klassi. Selleks, et tegemist oleks Nystrom'i raamatus toodud näitega sarnase lahendusega, peaks tõlke leidmise loogika eraldama GUIController klassist ja selle kohta on lõputöö autor jätnud ka kommentaari vastavasse klassi. Järgmine programmeerimismuster on aga nii levinud, et seda ei pidanud lõputöö autor isegi ise implementeerima.

3.4.4. Jälgija muster

Nystromi sõnul on jälgija muster (ingl „observer pattern“) üks levimumaid programmeerimismustreid – lausa nii levinud, et mitmetesse programmeerimiskeeltesse on see juba vaikimisi sisse ehitatud. Üks neist keeltest on ka C#. Nystromi sõnul on seda mustrit hea kasutada siis, kui mingi sündmuse peale peaks kuskil mujal mingi seotud tegevus käivituma, kuid koodis peaksid need kaks kohta sõltumatud olema. Jälgija mustri puhul saabki rakenduse mingi osa teatada mingist sündmusest kõigile kuulajatele ning samas pole tähtis kas neid kuulajaid on null või rohkem [7].

SOFIT Level Editor projektis on kasutatud C# programmeerimiskeelde sisse ehitatud jälgija mustrit. Ühe näitena võib tuua olukorra, kus ortograafilise kaamera suurus muutub. Sellest sündmusest teavitatakse seejärel kõiki kuulajataid. Üks neist kuulajatest on BlueprintController klass, mis selle peale käivitab ruudustike kuvamise meetodi. Nii on rakenduses alati nähtaval õige suurusega ruudustikud.

Teise näitena võib tuua plaani joonestamise režiimi, kus kutsutakse iga plaani muutuse (näiteks joone joonistamine, selle muutmine, ukse või akna paigutamine, muutmine või kustutamine

jne) peale välja meetod WallsGeometryChanged(), mis kuulutab sellest sündmusest kõigile kuulajatele. Selle eelis on see, et sama sündmust saavad välja kutsuda mitmed erinevad kohad ja sündmuse kuulaja ei pea neid kõiki eraldi kuulama. Siinkohal võib välja tuua, et kuna eelnevalt mainitud plaani muutused on implementeeritud käsumustrit kasutades, on selle spetsiifilise sündmuse kuulutamine väga lihtne. Järgmine alapeatükk kirjeldabki käsumustrit ning miks seda rakenduses kasutati.

3.4.5. Käsumuster

Nystromi näitel on käsumustri (ingl „command pattern") idee selles, et eksisteerib abstraktne ülemklass (näiteks Käsk või Command), millel on vähemalt üks abstraktne meetod käsu käivitamiseks ja vajadusel ka sisendparameetrid sellele meetodile. Iga käsk on selle abstraktse klassi alamklass ning igaüks implementeerib käsu käivitamisel tehtavad tegevused ise. Nii on kõik käsud sama tüüpi ja iga käsu loogika on isoleeritud käsu sisse. Nystrom ütleb veel, et kõige tuntum kasutuskoht sellel mustril on käskude tagasi- ja edasikeeramine [7].

Käskude tagasi- ja edasikeeramine oligi põhiline ajend, miks SOFIT Level Editor projektis kasutati käsumustrit. Rakenduses on abstraktne klass Action (vt Koodiplokk 3), milles on defineeritud neli meetodit. Iga ruumiplaani joonestamise või objektide paigutamisega seotud tegevuse jaoks on implementeeritud oma käsuklass ning tegevuse enda loogika on viidud sellesse klassi.

```
public abstract class Action
{

    public abstract string Name { get; }

    public abstract bool Execute();
    public abstract void Undo();
    public abstract void Redo();

    public abstract void Discard();
}
```

Koodiplokk 3. Käsumustri implementatsioon rakenduses SOFIT Tasemeredaktor.

Iga käsk salvestatakse pärast õnnestunud käivitumist (meetod Execute() tagastab tõeväärtuse tõene, kui käsu käivitumine õnnestus) pinusse ning kui kasutaja soovib viimati tehtud tegevust tagasi võtta, käivitatakse meetod Undo(), mis selle täide viib (ja sarnaselt meetod Redo() teeb tegevuse uuesti). Meetod Discard() ei ole funktsionaalsuse mõttes vajalik, see on loodud optimeerimise jaoks. Kui mingid pinu käsud visatakse minema (näiteks kui kasutaja võttis mitu tehtud käsku tagasi ning lisas uue käsu), siis kutsutakse iga minema visatud käsu puhul välja Discard() meetod, mis kustutab vajadusel üleliigseid objekte.

3.5. Kasutajamugavus

Tarkvara loomise puhul pannakse üldjuhul esmalt kirja funktsionaalsed nõuded. Need kirjeldavad ära funktsionaalsuse, mida loodav rakendus toetab. Selleks, et rakendus oleks lõppkasutajale kasulik, peab rakenduse funktsionaalsus toetama tema poolt soovitud ülesande lahendamist. Maailm on liikunud viimase kolme kümnendi jooksul käsurealt graafiliste liideste peale ning käskude tähthaaval sisestamise asemel saavad kasutajad erinevaid sisendseadmeid (näiteks arvutihiir või graafikalaud[26]) kasutades valida visuaalselt kasutajaliidesest vajalikke funktsionaalsuseid. Sellega on üha enam muutunud päevakohaseks inimese ja arvuti vaheline interaktsioon ning järjest rohkem tuleb tähelepanu pöörata mitte ainult sellele, mida rakendus teha oskab, vaid kuidas kasutaja seda kõige efektiivsemalt ära saab kasutada.

Kashfi, Nilsson ja Feldt [9] arutlevad, et kõrgekvaliteedilise tarkvara loomisel ei piisa tänapäeval enam rakendusse funktsionaalsuse lisamisest, vaid tähelepanu peab pöörama ka teistele mittefunktsionaalsetele aspektidele nagu testitavus ja kasutajamugavus. Kasutajamugavust on keeruline defineerida, aga Klimczak [10] toob enda raamatus paralleeliks pillimängimise, kus algaja muusik peab keskenduma suure osa oma tähelepanust sellele, et näpud õiges kohas oleks ja pillist õiged noodid välja tuleks. Kui aga muusik on juba osavam, liigub Klimczaki sõnul tähelepanu üha enam muusika mängimisele kuni selleni, kus pill on juba piltlikult öeldes muusiku käepikendus ja muusik saab täielikult keskenduda muusika loomisele. Tema sõnul on tarkvaraarenduses sarnane eesmärk - tarkvara disainija eesmärk on luua sellised interaktsiooni mehhanismid kasutaja ja rakenduse vahel, et pärast lühikest tutvumist programmiga suudab kasutaja seda mugavalt kasutada. Kui kasutaja suudab rakendust kasutades jõuda sellisesse meeleseisundisse, kus kogu tema tähelepanu saab olla käesoleva ülesande lahendamisel ja teda ei sega pausid (näiteks õige nupu leidmine) või ootamatused rakenduse käitumises, siis võib rääkida hästi disainitud interaktsioonidest [10].

Käesolev peatükk on jagatud kaheks alapeatükiks. Esmalt on kirjeldatud teiste autorite tähelepanekud CAD rakenduste kasutajaliideste kohta ning seejärel on välja toodud mõned head tavad kasutajaliideste disainis ja nende kasutused rakenduses SOFIT Tasemeredaktor.

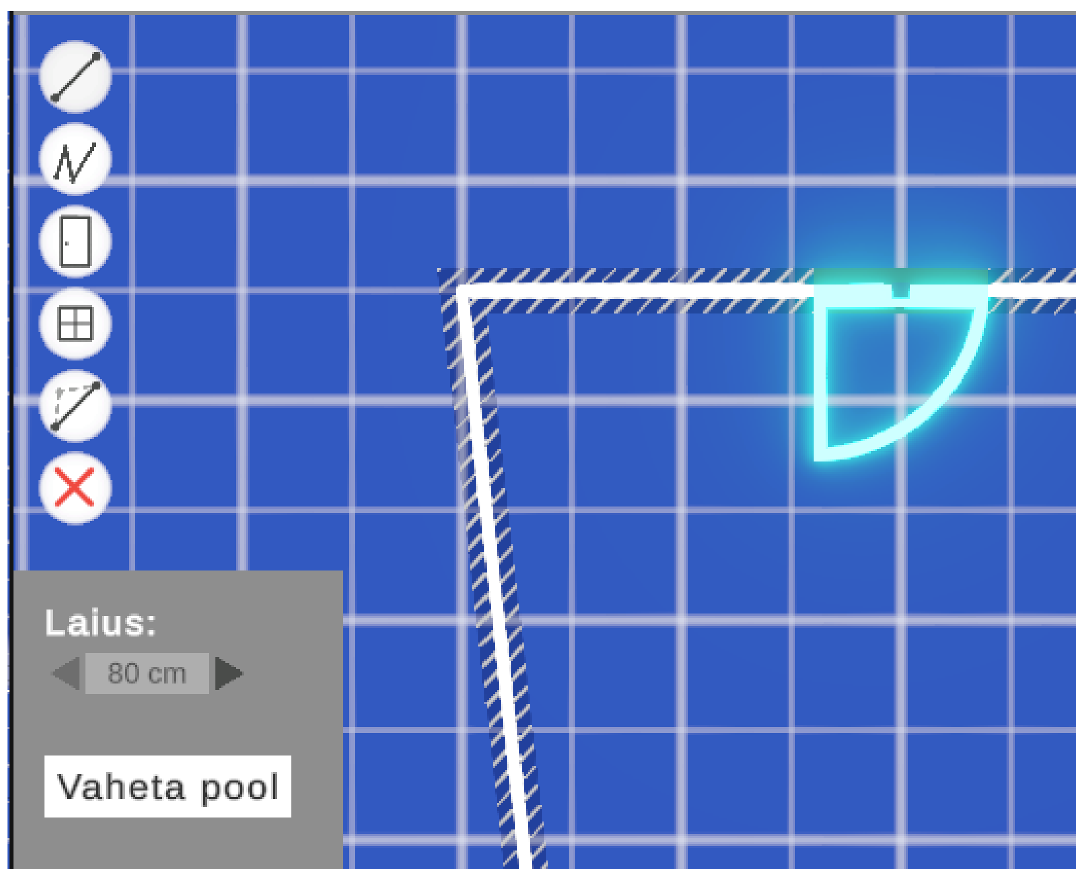
[26] Arvuti sisendseade, millega kasutaja saab pliiatsilaadse seadmega (ingl „stylus”) graafikalaua ekraani peale kirjutada või joonistada nii, et see joonistus jõuab arvutiekraanile.

3.5.1. Tähelepanekud CAD rakenduste kasutajaliidestes

Bhavnani, Flemming, Forsythe, Garrett, Shaw ja Tsai toovad enda empiirilises uuringus välja, et CAD rakenduse kasutajad teevad sageli korduvaid operatsioone ning kui nupud või funktsioonid nende tegemiseks on mitme kliki kaugusel, kulutab see märkimisväärselt kasutajate aega [11].

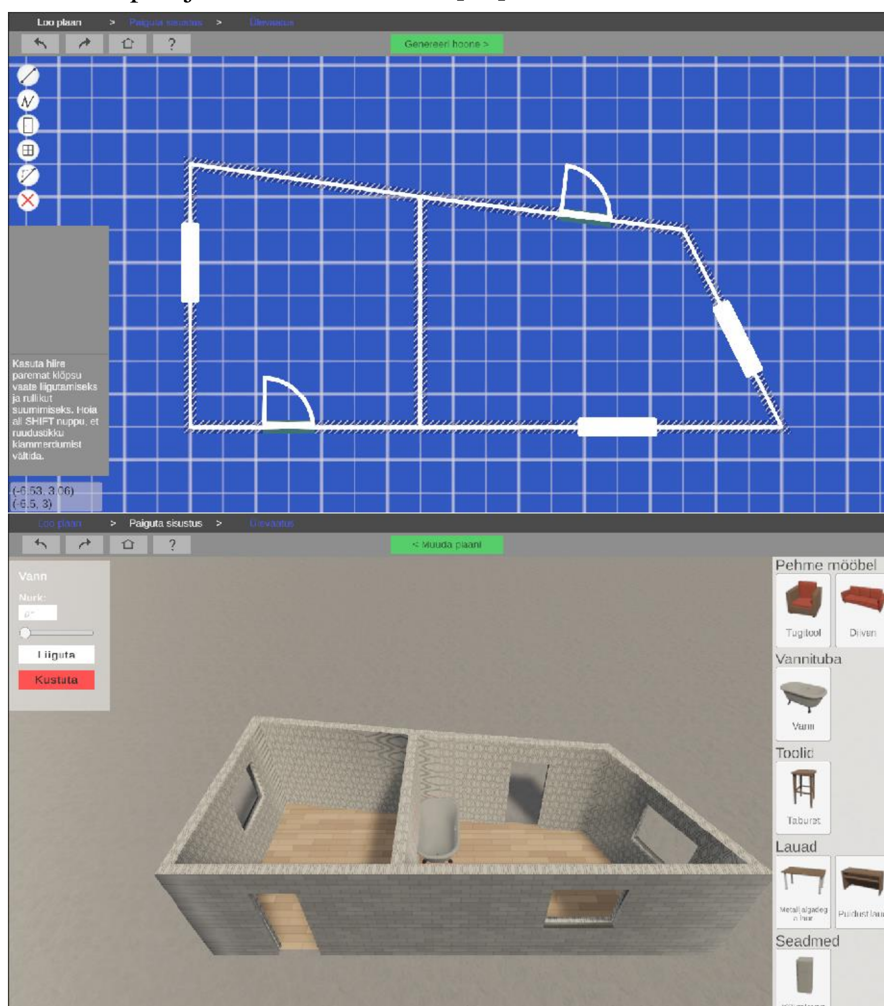
Kuigi ülalmainitud uuring on avaldatud juba rohkem kui 25 aastat tagasi, leiab töö autor, et selles olevad leiud on päevakohased ka praegu. Ülaltoodud tähelepaneku põhjal saab järeldada, et tihedamini kasutatavad funktsionaalsused peaksid olema lihtsasti leitavad, et hoida kasutaja aega kokku. Selle saavutamiseks on vajalik kaardistada kasutajate tegevused, need sorteerida kasutustiheduse järgi ning leida üles funktsionaalsused, mida kasutatakse kõige rohkem.

SOFIT Tasemeredaktor rakenduse puhul pole piisavalt funktsionaalsust, et see probleemiks oleks, kuid sellegipoolest on seda kasutajaliidese arendamisel arvesse võetud. Põhifunktsionaalsused, nagu plaani joonestamise tööriistad, on lisatud eraldi menüüna ekraani äärde, kuid vähemkasutatavad funktsionaalsused, nagu näiteks ukse avanemise suuna või ukse laiuse muutmine, on nähtavad vaid siis, kui mõni uks valitud on (vt Joonis 23).



Joonis 23. Pidevalt nähtavad põhifunktsionaalsuse nupud (ümmargused) ning ainult vajadusel kuvatavad ukse laiuse ja poole vahetuse nupud.

Lisaks märgivad Bhavnani jt, et kasutajatel on tendents ära õppida vaid vähesed käsud ja ainult neid kasutada, isegi kui eksisteerivad keerulisemad käsud, mis oleksid tunduvalt efektiivsemad [11]. Lee, Eastman, Taunk ja Ho [12] leidsid samuti, et CAD rakenduste kasutajaliidestes on tihti sadu menüüelemente, mis koos erinevate sätetega kasvatavad võimalike operatsioonide arvu suuremaks kui rakenduse kasutaja suudab meeles. Sarnase mõtte esitab ka Klimczak ning pakub välja ka lahenduse. Tema sõnul peaks igal hetkel olema rakenduses kuvatud vaid kontekstiga seotud nupud ja funktsionaalsused [10].



Joonis 24. Erinevad menüüpaneelid erinevates režiimides.

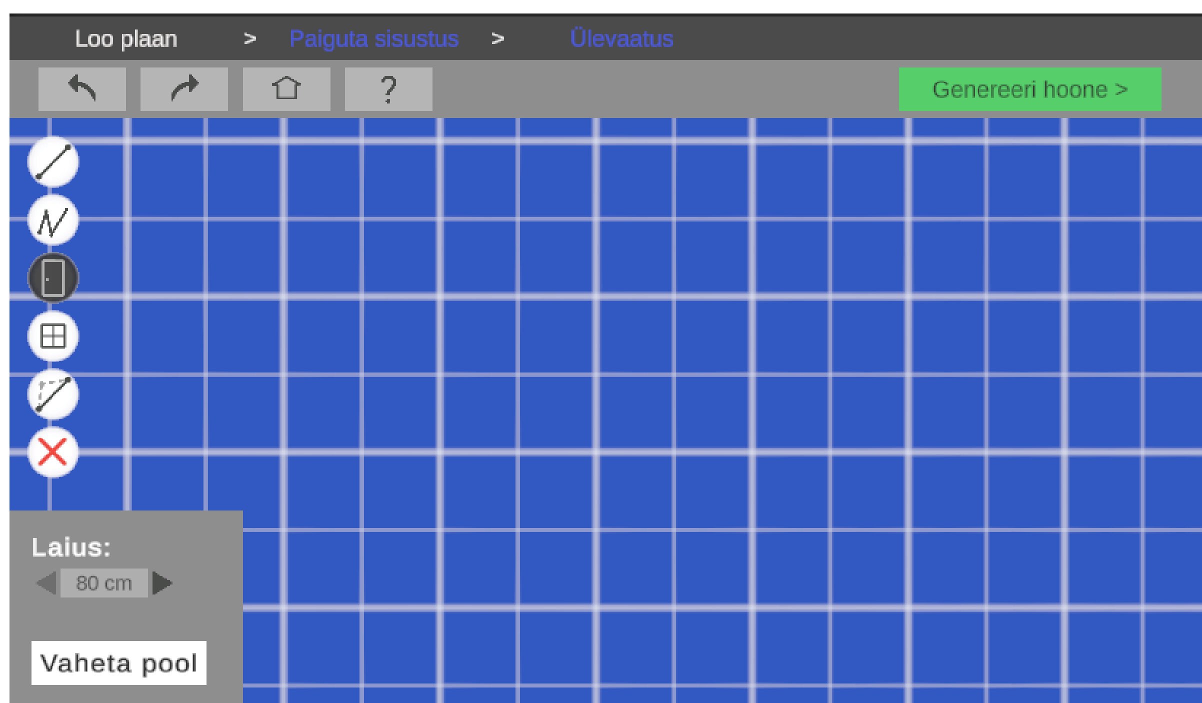
Klimczak'i esitatud soovitust on kasutatud ka SOFIT Tasemeredaktor rakenduses. Plaani joonestamise ja objektide paigutamise režiimides on kuvatud vaid vastava režiimiga seotud menüüaknad. Üle rakenduse töötavad funktsionaalsused on viidud üldisele menüüribale, mis on kuvatud läbi terve rakenduse (vt Joonis 24). Järgmises alapeatükis on toodud välja üldisemaid soovitusi kasutajaliidese disainiks.

3.5.2. Head tavad kasutajaliideseid disainides

Ideaalset kasutajaliidest ei ole võimalik disainida, sest kasutajad ja nende soovid on nii erinevad. Sellegipoolest on mitmeid tavasid, millest kinni pidades saab luua kasutajaliidese, mida suurem osa kasutajatest võiks intuitiivseks pidada. Kuna neid tavasid on liiga palju, et neist kõigist rääkida, on välja toodud vaid mõned üksikud, mis reaalselt rakenduse kasutajaliidese disaini mõjutasid.

Üks põhiline rakenduse komponente on nupp. Kasutaja kasutab nuppe, et mingisugune funktsionaalsus käivitada, näiteks aktiveerida sobiv joonestamisrežiim või seina paksust muuta. Peaaegu kõik funktsionaalsused, mis SOFIT Tasemeredaktoris on implementeeritud, on käivitatavad mingi nupuga. Milline peaks aga olema üks nupp?

Cooper, Reimann, Cronin ja Noessel toovad enda raamatus välja, et nupu eesmärk peaks olema nupu peale kirjutatud, siis on nupust lihtsam aru saada. Nende sõnul saab algaja kasutaja paremini aru rakenduse võimekusest ja olemasolevatest võimalustest, kui nuppude peal on tekstid, mitte ikoonid. Samas juba kogenenuma kasutaja puhul ei määra see enam nii suurt rolli, kuna ta juba teab, kus otsitav nupp on [13]. Samas toovad Cooper jt sellele ka vastuargumendi – mõndades rakendustes on tähtis võimalikult palju ekraanipinda hoida menüüdest vabana ja sel juhul on kasulikum tekstide asemel ikoone kasutada, sest need võtavad vähem ruumi [13].



Joonis 25. Erinevad nupud rakenduses.

Täpselt selline olukord on autori arvates ka SOFIT Tasemeredaktor rakenduses. Kasutajal on mugavam hoone plaani joonestada või objekte maailma paigutada kui ta päralt on võimalikult

suur ala. Lisaks on (vähemalt lõputöö esitamise hetkel) rakenduses veel üsna vähe funktsionaalsust, seega peaks olemasolevate nuppude funktsionaalsused kiiresti kasutajale meelde jääma. Sellegipoolest on rakenduses ka nuppe, millel on ikooni asemel tekst. Üks näide sellest on ukse avanemise poole vahetuse nupp. Sellele oleks raske üheselt mõistetavat ikooni leida ning samas on see juba üsna laia paneeli peal, mistõttu saab ka nupp selle võrra suurem olla (vt Joonis 25).

Cooper jt toovad leevendava meetmena välja ka hüpikvihjete kasutamise, mis lihtsustavad algaja kasutaja rakenduse õppimist. Nad soovitavad lisada ka kiirklahvi olemasolul selle väärtuse nuppude lähedale, et õpetada kasutajale nende kasutamist [13].

Seda soovitust on kasutatud ka SOFIT Tasemeredaktor rakenduses (vt Joonis 26). Selleks, et see funktsionaalsus töötaks rakenduses igal pool sarnaselt, loodi klass ToolTipData kahe sõne tüüpi andmeväljaga hotKey ja helpTextCode. Rakendus kontrollib pidevalt, kas kursor on mõne GUI elemendi kohal, millel on komponendina lisatud klass ToolTipData ning kui on, siis kuvatakse väikse viivitusega (viivituse pikkust saab määrata rakenduse sättefailides) hüpikvihje paneel kursori lähedal. Kiirklahvide puhul on veel vaja otsustada, milliseid funktsionaalsuseid peaks saama kiirklahviga aktiveerida ja milliseid mitte.

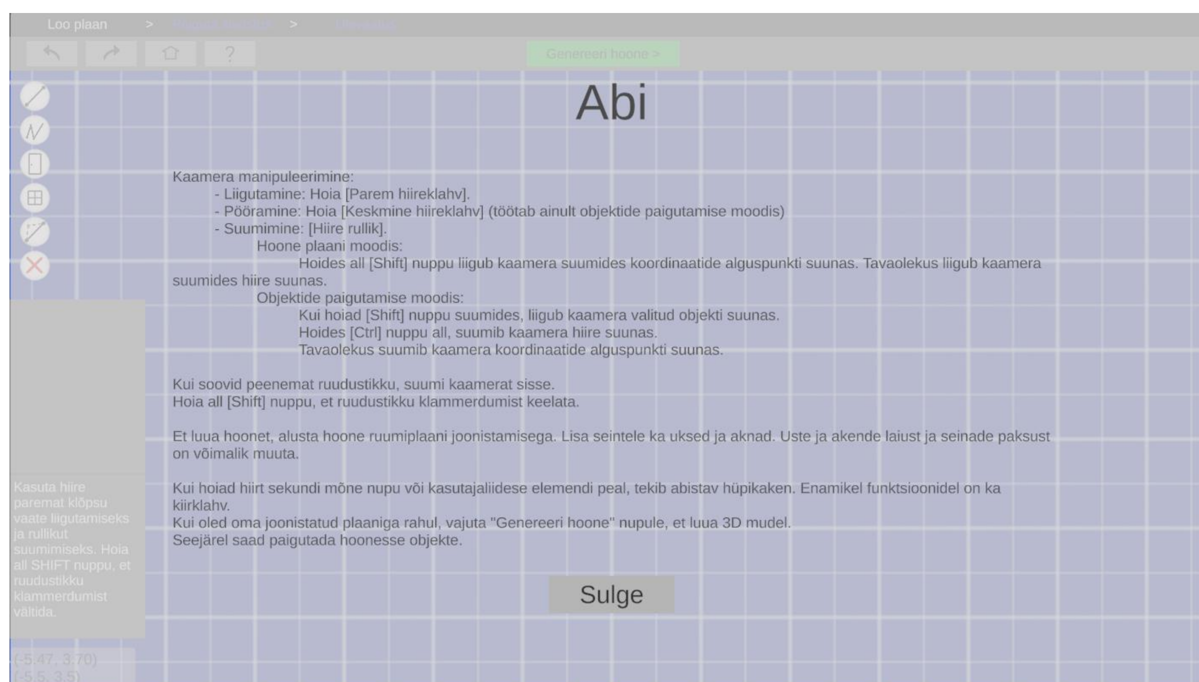
Joonis 26. Hüpikvihje kiirklahvi ja lühikese kirjeldusega rakenduses.

Cooper jt [13] soovitavad funktsionaalsused kolmeks jagada ja seejärel otsustada nende gruppide järgi, kas funktsionaalsustele lisada kiirklahv.

1. Esimene grupp, kus on iga tüüpi kasutaja igapäevaselt kasutatavad funktsionaalsused, peaksid alati olema aktiveeritavad ka kiirklahvidega.
2. Teine grupp on need funktsionaalsused, mida ükski kasutaja ei kasuta igapäevaselt. Nende puhul ei tohiks lisada kiirklahve.
3. Kolmas grupp on funktsionaalsused, mis ei kuulu esimesse ega teisse gruppi. Nende puhul peaks ükshaaval hindama, kas funktsionaalsus vajab kiirklahvi.

Lisaks mainivad nad kolmanda grupi puhul, et funktsionaalsused, mida kasutatakse tihedamini, peaksid olema lihtsamate kiirklahvidega (näiteks üksik klahvivajutus nagu W või F1) ja need, mida kasutatakse harvemini, võivad vajadusel olla keerulistemate klahvikombinatsioonidega (näiteks Ctrl + Shift + C). Veel toovad nad välja, et üldkasutatavate funktsioonide

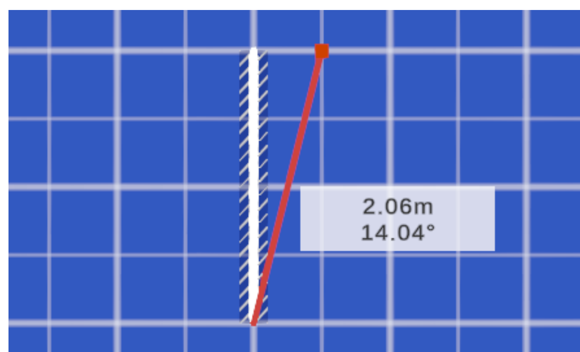
(kopeerimine, kleepimine, tagasi võtmine jne) puhul peaks kiirklahvid olema sarnased üldkasutatavatele tavadele [13].



Joonis 27. Abipaneel, mis avaneb automaatselt rakenduse käivitumisel.

Kuna arendatav rakenduse skoop on minimaalne töötav toode, on olemasolevat funktsionaalsust vähe. Seetõttu hindas töö autor, et suurem osa olemasolevatest nuppudest peaksid olema esimeses või kolmandas grupis ning lisas neile kiirklahviga aktiveerimise võimekuse.

Viimaks võib veel välja tuua Klimczak'i soovituse, et kui kasutajale on vaja näidata teateid, siis tuleb võimalusel vältida töövoogu rikkuvaid meetodeid. Tema sõnul peaks kinnitust vajavad hüpikmenüüd ja kasutaja tähelepanu püüdvad helisignaalid hoidma vaid kriitiliste probleemide jaoks [10].



Joonis 28. Joone mittesobival olekul kuvatakse joon punasena.

SOFIT Level Editor rakenduses lõputöö esitamise hetkel ühtegi hüpikakent pole (kui mitte arvesse võtta rakenduse avamisel kuvatavat abipaneeli (vt Joonis 27)). Kasutaja tähelepanu nõudvad probleemid kuvatakse kasutajale jooksvalt ning töövoogu mitte takistavalt. Näiteks on rakenduses keelatud alla 25-kraadiste nurkadega seinte loomine. Kui kasutaja üritab sellist seina paigutada, kuvatakse paigutatav sein teistsuguse välimusega ning hiireklikiga seina salvestamine ei õnnestu (vt Joonis 28).

4. Kasutajatestimine

SOFIT Tasemeredaktor rakenduse üks mittefunktsionaalseid nõudeid oli hea kasutajamugavus. Kuigi seda on raske objektiivselt hinnata, saab seda kaudselt hinnata läbi kasutajamugavuse testimise[27]. Kasutajamugavuse testimisel on vaja testijaid, luua test-stsenaarium ning hiljem tõlgendada tulemusi. Allpool on toodud kolmes alapeatükis välja testijate valik, testi stsenaarium ja lõpuks testimise tulemused ja nende põhjal tehtud soovitused rakenduse edaspidiseks arenduseks.

4.1. Testijad

Testijateks valiti mittejuhuslikult kolm erineva arvutikasutamise taustaga inimest erinevatest valdkondadest. Testija A on 31-aastane ligi 10-aastase kogemusega tarkvaraarendaja, kellel on väga pikk kogemus arvutiga töötamisel. Testija B on 27-aastane bioanalüütik, kes kasutab arvutit igapäevaselt, kuid lihtsamate toimingute jaoks (veebi brausimine). Testija C on 53-aastane projekteerija, kes on vähese igapäevase arvutikasutamise oskusega, kuid on kasutanud mitmeid CAD rakendusi oma igapäevatöös.

Valim on väike põhjusel, et tegemist on minimaalse töötava tootega ja esialgu on vaja veenduda, kas valitud disainisuunad on õiged või valed. Ükski testija ei olnud enne testimist SOFIT Tasemeredaktor rakendust kasutanud.

4.2. Testi stsenaarium

Testijate ülesandeks oli taasluua neile etteantud visand ühest keskmise keerukusega hoonest (vt Lisa B: Kasutajatestimisega seotud failid), mida luues peaks testija kasutama enamikke valminud rakenduse funktsionaalsuseid. Abimaterjale peale rakenduses olevate abivahendite (abipaneel, hüpikvihjed) töö autor neile ei võimaldanud ja stsenaariumi täitmisel oli tema roll passiivne. Sellega oli simuleeritud olukord, kus kasutaja peab ilma välise abita rakenduse kasutamisega hakkama saama. Lisaks loob see olukorra, kus testijad ei ole kuidagi suunatud rakendust õigesti kasutama, seega on neil vabad käed rakendust valesti kasutada. Selle põhjal saab kasutajaliidese nõrgad kohad üles leida.

[27] https://www.guru99.com/usability-testing-tutorial.html

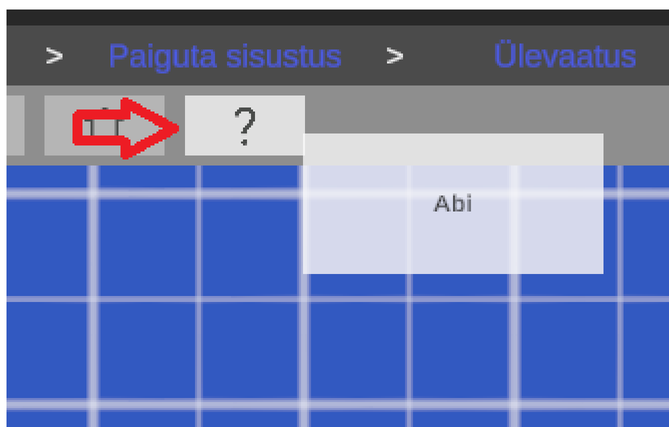
Test-stsenaariumi toimumise hetkel oli töö autor vaatleja ning pani kirja märkused rakenduse kasutamise kohta. Pärast testimis-sessiooni viidi läbi ka lühike intervjuu kasutajakogemuse kohta, milles kaks eelnevalt määratud küsimust:

1. Mis häiris Sind rakendust kasutades?
2. Mis meeldis Sulle rakendust kasutades?

Lisaks neile kahele küsimusele küsis töö autor veel mõningaid täpsustavaid küsimusi, et aru saada testijate poolt tehtud (või mitte tehtud) tegevustest. Järgnevalt on välja toodud testimise tulemused ja soovitused edasisteks täiendusteks.

4.3. Testimise tulemused

Kõik kolm testijat said etteantud katseülesandega hakkama. Keskmiselt võttis plaani joonestamine aega 16.7 minutit ning objektide paigutamine 5.7 minutit (vt Tabel 1). Plaani joonestamise aja sisse kulus ka rakenduse käivitumisel avaneva abipaneeliga tutvumine. Kaks testijat kolmest tõid pärast testülesande sooritamist ka välja, et abipaneel oli neile suureks abiks, kuid lõputöö autorile jäi testsooritusi jälgides silma, et kõik kolm testijat veetsid vähemalt 10 sekundit, et abipaneeli uuesti avavat nuppu üles leida. Sellest võib järeldada, et selle nupu ikoon ei ole hästi valitud (vt Joonis 29).



Joonis 29. Abipaneeli avav nupp rakenduses.

Tabel 1. Testülesande sooritamiseks kulunud aeg.

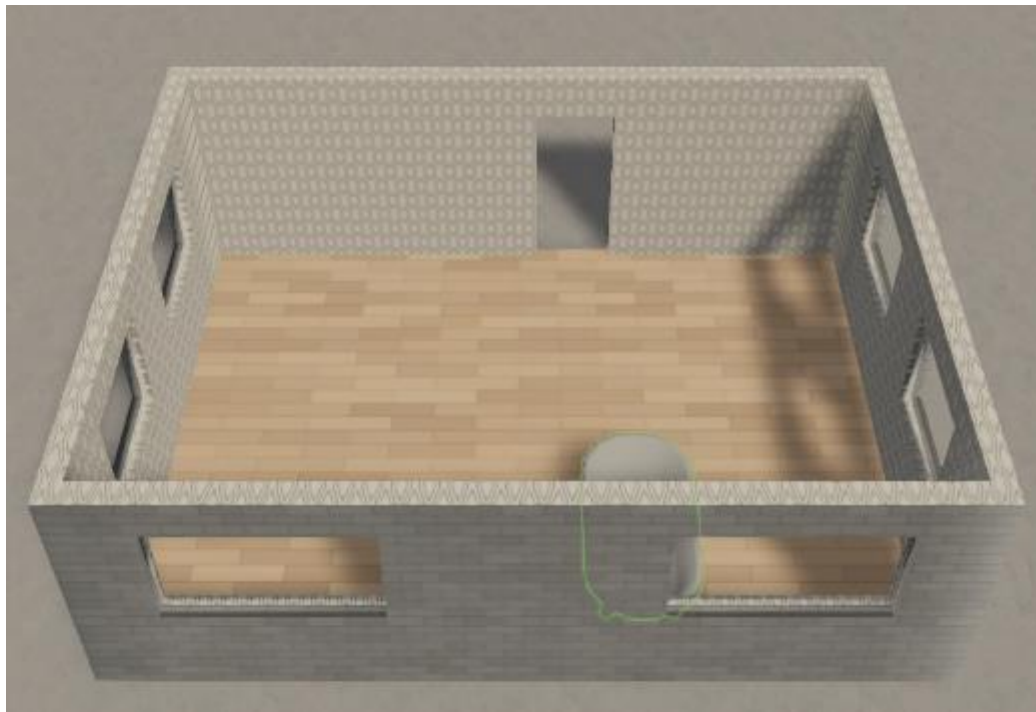
Testija	Plaani joonestamiseks kulunud aeg (minutites).	Objektide paigutamiseks kulunud aeg (minutites).	Kokku kulunud aeg (minutites).
Testija A	15	5	20
Testija B	15	5	20
Testija C	20	7	27
Keskmine	**16.7**	**5.7**	**22.3**

Peatükis 3.5.2 olid välja toodud põhjused, miks peaaegu kõik rakenduse nupud ja funktsionaalsused on ka kiirklahvidega aktiveeritavad. Kolmest testijast kaks aga ei kasutanud ühtegi olemasolevat kiirklahvi (kuigi testija C üritas ühe korra kasutada kiirklahvi, mida rakenduses ei oldud implementeeritud – klahvikombinatsiooni Ctrl + C joone kopeerimiseks).

Funktsionaalsus, mida ükski testija ei kasutanud, oli mitme joone mood, kus iga klikiga saab järgmise seina lisada. Kõik testijad nägid esimest nuppu rakenduses (üksiku joone lisamise mood) ning hakkasid seda kasutades kohe plaani joonestama. Alles pärast seinte paikapanekut avastati, et on olemas ka mugavam mood joonte lisamiseks. Samas küsimuse peale, miks nad seda moodi ei kasutanud, ütlesid kõik kolm, et kui nad peaks seda rakendust veel kasutama, siis nad tõenäoliselt kasutaks ka seda joonestamismoodi.

Üks probleem, mida kõik testijad välja tõid, oli objektide paigutamise moodis kaamera liigutamise kohmakus. Sellega nõustub ka töö autor – selle funktsionaalsuse viimistlemisega oleks võinud rohkem jõuda tegeleda.

Testijad A ja B tõid mõlemad välja soovitusena selle, et objekte paigutades võiks paigutust varjavad seinad läbipaistvaks muuta, et oleks paremini aru saada, kuhu objekt paigutub (vt Joonis 30). Veel tõi testija B välja, et talle meeldiks, kui objekte saaks 90° kaupa pöörata. Iganädalaselt CAD rakendusi kasutav testija C tundis aga puudust sellest, et seinte pikkuseid saaks määrata klaviatuuri kasutades ning et mitmel seinal saaks korraga paksust muuta.



Joonis 30. Sein blokeerib vanni mudeli seda paigutades.

Kokkuvõtvalt võib välja tuua soovitused olemasolevate funktsionaalsuste täiendusteks:

1. Objekti paigutamise moodis kaamera liigutamine ja pööramine võiks toimida sujuvamalt ja intuitiivsemalt.
2. Objektide paigutamise moodis võiks objekte peitvad seinad läbipaistvaks muuta.
3. Abipaneeli nupu ikoon tuleks ära muuta arusaadavamaks ikooniks või tekstiks „Abi“.
4. Plaani elementide muutmine võiks toimida ka mitme objekti korraga valimise puhul.
5. Seinte pikkuseid võiks saada määrata klaviatuuri kasutades.

Töö autor nõustub kõigi ettepanekutega.

Viimaks võib välja tuua ka positiivsed kommentaarid rakenduse kohta. Kõigile kolmele testijale meeldisid abipaneelid joonte paigutamisel, mis näitavad joone pikkust ja nurka (vt peatükis 3.2.2 Joonis 7). Lisaks meeldisid kõigile kolmele testijale ka uste ja akende paigutamise mugavus. Testijad B ja C tõid lisaks näitena välja teised rakendused (testija B puhul arvutimänguseeria Sims[28] ja testija C puhul rakendus AutoCAD[29]), kus uste ja akende paigutamine on ebamugavam kui rakenduses SOFIT Tasemeredaktor. Testijale A meeldis rakenduse puhul see, et tema väikse vaeva peale tekkis lihtsatest joontest kolmemõõtmeline mudel.

[28] https://www.ea.com/games/the-sims/the-sims-4
[29] https://www.autodesk.com/products/autocad/overview

5. Kokkuvõte

Käesoleva töö eesmärk oli luua minimaalselt töötav toode SOFIT Tasemeredaktor, millega oleks kasutajal võimalik luua kolmemõõtmelisi hooneid ja neisse objekte paigutada. Lõputöö tulemusena valmis rakendus, milles on võimalik soovitava hoone ruumiplaan joonestada ning sellest ühe nupuvajutusega 3D-mudel genereerida.

Töö alguses tutvustati kolme sarnast rakendust ning toodi välja iga rakenduse puhul selle positiivsed ja negatiivsed küljed, kui eesmärk oleks neid kasutada SOFIT Tasemeredaktor asemel. Seejärel tutvustati rakenduse arenduseks kasutatud tehnoloogiaid.

Töö autor jagas loodava rakenduse kolme funktsionaalsesse ossa. Esimeses osas loob kasutaja soovitud hoonest plaani, teises osas genereeritakse sellest plaanist kolmemõõtmeline võrestik ning kolmandas osas saab kasutaja loodud hoone mudelisse objekte (näiteks mööblit) paigutada. Esimese osa – hoone joonestamise režiimi – arenduse käigus selgus, et esimeseks lahenduseks valitud mänguobjektide põhine ruudustik ei ole hea lahendus. Seejärel kirjeldas töö autor paremat lähenemist kasutades varjutajat.

Kolmemõõtmelise võrestiku genereerimine oli tehniliselt kõige keerulisem ülesanne ning selle lahenduse kirjeldus on ka lõviosa tööst. Selles on kirjeldatud, kuidas leida eelnevalt joonestatud plaani pealt kõik ruumid ning neile ruumidele vastavad sise- ja välisseinade võrestikud. Seejärel toodi välja põhjused, miks sama lähenemist ei saa kasutada põrandate ja lagede võrestike loomisel ja toodi välja kolm alternatiivset meetodit ning nende plussid ja miinused. Seejärel kirjeldati valituks osutunud Delaunay algoritmi reaalset implementatsiooni, mis põhines Sloan'i kahel artiklil.

Järgmisena tõi autor välja ka kasutatud programmeerimismustrid: olekumuster, singleton'i muster, teenuseotsija muster, jälgija muster ja käsumuster. Implementatsiooni peatüki lõpetas lühike arutelu kasutajamugavuse teemal ning ka mõned näited selle juurutamisest rakenduses.

Loodud rakendust testis ka kolm erineva arvutikasutamise oskusega testijat. Töö autor luges töö õnnestunuks, kuna kõigil testijatel õnnestus etteantud katseülesanne mõistliku aja jooksul ära lahendada ilma abi küsimata. Testimisest tulid välja ka mõned puudujäägid rakenduses, kuid kuna tegemist oli minimaalse töötava tootega, siis oli see oodatav. Viimaks tõi töö autor antud tagasiside põhjal välja viis soovitust rakenduse edasiseks arenduseks.

Töö autor tänab kõiki testijaid, kõiki kolme lõputöö juhendajat, Raimond Tunnelit ning Ingmari Trumpi Criffinist.

Kasutatud kirjandus

[1] P. J. Frey and P.-L. George, *Mesh Generation: Application to Finite Elements*, 2nd edition. Great Britain and USA: ISTE Ltd and John Wiley and Sons, Inc, 2008.

[2] S.-W. Cheng, T. K. Dey, and J. R. Shewchuk, *Delaunay Mesh Generation*. United States of America: CRC Press, 2013.

[3] S. W. Sloan, 'A fast algorithm for constructing Delaunay triangulations in the plane', *Adv. Eng. Softw.*, vol. 9, no. 1, pp. 34–55, 1987, doi: 10.1016/0141-1195(87)90043-X.

[4] S. W. Sloan, 'A fast algorithm for generating constrained delaunay triangulations', *Comput. Struct.*, vol. 47, no. 3, pp. 441–450, 1993, doi: 10.1016/0045-7949(93)90239-A.

[5] A.K. Cline and R.L. Renka, 'A storage-efficient method for construction of a Thiessen triangulation', *Rocky Mt. J. Math.*, vol. 14, no. 1, pp. 119–140, Mar. 1984, doi: 10.1216/RMJ-1984-14-1-119.

[6] E. Gamma, Ed., *Design patterns: elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995.

[7] R. Nystrom, *Game programming patterns*. s.l.: genever benning, 2014.

[8] J. Wang, *Formal Methods in Computer Science*. CRC Press, 2019.

[9] P. Kashfi, A. Nilsson, and R. Feldt, 'Integrating User eXperience practices into software development processes: implications of the UX characteristics', *PeerJ Comput. Sci.*, vol. 3, p. e130, Oct. 2017, doi: 10.7717/peerj-cs.130.

[10] E. Klimczak, *Design for software: a playbook for developers*. Hoboken, N.J: Wiley, 2013.

[11] S. K. Bhavnani, U. Flemming, D. E. Forsythe, J. H. Garrett, D. S. Shaw, and A. Tsai, 'CAD usage in an architectural office: from observations to active assistance', *Autom. Constr.*, vol. 5, no. 3, pp. 243–255, Sep. 1996, doi: 10.1016/0926-5805(96)00149-5.

[12] G. Lee, C. M. Eastman, T. Taunk, and C.-H. Ho, 'Usability principles and best practices for the user interface design of complex 3D architectural design and engineering tools', *Int. J. Hum.-Comput. Stud.*, vol. 68, no. 1–2, pp. 90–104, Jan. 2010, doi: 10.1016/j.ijhcs.2009.10.001.

[13] A. Cooper, D. Cronin, C. Noessel, and R. Reimann, *About face: the essentials of interaction design*. Indianapolis, IN: John Wiley and Sons, 2014.

Lisad

A. Tööga seotud failid

Tööga kaasas olevas ZIP failis on kõik autori poolt loodud spraidid. Kasutatud mudelid võimaldas töö autorile Criffin ja neid tööga kaasas ei ole. Kaasas on ka mõned kuvatõmmised rakenduse erinevatest osadest ja lühike video rakenduse kasutamisest. Lisaks on ZIP-failis autori märkmed kasutajatestimise kohta. Faili struktuur on järgnev:

- Kaust „Kasutajatestimine“ – sisaldab kasutajatestimisega seotud faile
  - markmed.txt – vabas vormis autori märkmed kasutajatestimise kohta.
  - ruumiplaan.png – teststsenaariumi kirjeldav joonis.
  - a1.png – testija A loodud hoone plaan.
  - a2.png – testija A loodud hoone lõpptulemus.
  - b1.png – testija B loodud hoone plaan.
  - b2.png – testija B loodud hoone lõpptulemus.
  - c1.png – testija C loodud hoone plaan (testija C lõpptulemusest jäi pilt tegemata).
- Kaust „Rakendus_visuaalselt“ – sisaldab rakenduse kuvatõmmiseid ja demovideot.
- Kaust „Spraidid“ – sisaldab kõiki rakenduses kasutatud spraite.

B. Kasutajatestimisega seotud failid

Autor koostas joonise keskmise raskusastmega hoonest, mida rakenduse testijad pidid rakenduses taasloома. Pilt, mille põhjal nad seda tegid, on järgnev:

C. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Tanel Orumaa

annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose SOFIT Tasemeredaktor, mille juhendaja on Ulrich Norbisrath, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Tanel Orumaa

10.05.2022