TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Markus Saarniit

Voxel-World – maailm kuubikutest

Bakalaureusetöö (9 EAP)

Juhendaja: Raimond-Hendrik Tunnel, MSc

Tartu 2019

Voxel-World – maailm kuubikutest

Lühikokkuvõte:

Käesolevas bakalaureusetöös kirjeldatakse arvutimängu Voxel-World loomiseks kasutatud tehnoloogiaid ning erinevaid arvutigraafika võtteid nagu varju kaardistamine, võrestamine ja peegelduste renderdamine realistliku vee jaoks. Samuti kirjeldatakse teksti renderdamist OpenGL abil kasutades eelrasteriseeritud fonte.

Võtmesõnad:

Võrestamine, OpenGL, arvutigraafika, varjude renderdamine, vee renderdamine, vokslid, Java

CERCS:

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

P175: Informaatika, süsteemiteooria.

Voxel-World – a world of cubes

Abstract:

This Bachelor's thesis describes the technologies and different computer graphics methods used to implement Voxel-World computer game. Described methods include shadow mapping, meshing and reflections for realistic looking water. Also text rendering with OpenGL using previously rasterized fonts is described.

Keywords:

Meshing, OpenGL, computer graphics, rendering shadows, rendering water, voxels, Java

CERCS:

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

P175: Informatics, systems theory.

Sisukord

1. Sissejuhatus 5

2. Kasutatud tehnoloogiad 6

  2.1 OpenGL 6

  2.2 Lightweight Java Game Library 6

3. Maastik andmestruktuurina 8

  3.1 Maastiku tükid 8

  3.2 Võrestamine 10

    Naiivne võrestamise algoritm 12

    Praakiv võrestamise algoritm 12

    Agar võrestamise algoritm 13

4. Varjud 15

5. Vesi 17

  5.1 Refraktsioon ja peegeldus 17

  5.2 Lainetus 20

6. Kasutajaliides 22

  6.1 Teksti renderdamine 22

  6.2 Menüüd 25

7. Jõudlus 26

8. Kokkuvõte 28

9. Viidatud kirjandus 29

Lisad 30

  I. Terminid 30

  II. Failid 32

  III. Kasutusjuhend 33

IV. Teadaolevad vead 34

V. Litsents 35

1. Sissejuhatus

Töö eesmärgiks on luua arvutimänguga Minecraft visuaalselt sarnane arvutimäng Voxel-World. Minecrafti laadsete mängude põhiline omadus on ühikkuupidest koosnev maastik, mis on loodud täielikult arvuti poolt mängu käigus. Maastik on lõpmatu ning kasutaja saab maastikus asuvaid kuupe eemaldada ja maastikusse kuupe juurde lisada. Sellise rakenduse arenduse seisukohast kõige suuremad väljakutsed on maastiku andmemudeli hoidmine arvuti mälus ning selle põhjal kolmnurkadest koosneva võrestiku loomine. Käesolev töö kirjeldab Voxel-Worldi implementeerimise protsessi ning mängu tegemiseks kasutatud tehnoloogiaid.

Läbides ainet nimega Arvutigraafika tegime meeskonnaga projektiks mängu nimega Minecraft Rip-Off[1]. Kuna sellel mängul jäi puudu igasugune kasutajaliides, varjud ning vee renderdamine koos peegeldustega, siis seatakse nende funktsionaalsuste realiseerimine käeoleva töö põhiliste eesmärkide hulka. Sarnastest töödest tasub veel välja tuua Andreas Sepa bakalaureusetööd „Protseduuriline lõpmatu maastiku genereerimine“. Autor kirjeldas seal algoritmi, mille abil on võimalik arvutimänguga Minecraft sarnast maastikku tekitada. Samuti oli selles töös mõningaid ideid võrestamise ja maastiku andmestruktuuri teemadel [1].

Töö teises peatükis kirjeldatakse kasutatud tehnoloogiaid ning selgitatakse lühidalt nende ülesandeid. Seal edasi, kolmandas peatükis, antakse ülevaade lõpmatu maastiku hoidmisest arvuti mälus. Kolmanda peatüki teises pooles kirjeldatakse maastiku visualiseerimist ning erinevaid võrestamise algoritme.

Kuna varjude ja vee renderdamine on mahukad ja keerulised teemad arvutigraafikas, siis need funktsionaalsused on käsitletud eraldi vastavalt neljandas ja viiendas peatükis. Tuuakse välja mõned laialt levinud meetodid ning kirjeldatakse nende tööpõhimõtteid ning probleeme.

Käesoleva töö raames on mängule loodud ka lihtsakoelised menüüd, mida kirjeldatakse ka lühidalt kuuendas peatükis. Kuuenda peatüki põhiteemaks on aga teksti renderdamine kasutades OpenGL liidest. Viimane, seitsmes, peatükk räägib valminud rakenduse jõudlusest. Lisade all on olemas terminite selgitused ning kaasapandud failide kirjeldus.

---

[1] https://courses.cs.ut.ee/2017/cg/fall/Main/Project-MinecraftRipOff

2. Kasutatud tehnoloogiad

Arvutimängu loomiseks kasutasin Java programeerimikeelt, kuna see kasutab automaatset prügikoristust, mis vabastab programeerija manuaalsest mäluhaldusest. See võimaldab rohkem keskenduda funktsionaalsuse arendamisele ning jätta madalatasemeline mäluhaldus keskkonna enda hallata. Automaatse prügikoristuse puuduseks on ajutine võimsuse kadu, kui toimub prügikoristus. Kasutaja näeb seda mängu kaadrisageduse langusena[2]. Hoolimata sellest puudusest otsustasin valida siiski Java kuna tunnen seda programeerimiskeelt kõige paremini ning automaatne mäluhaldus võimaldab keskenduda rohkem rakenduse funktsionaalsuste arendamisele.

Kasutatud tehnoloogiatest tähtsaimad on 2D ja 3D vektorgraafika renderdamise rakenduseliides OpenGL ning Java ja OpenGL vahelist suhtlust korraldav teek Lightweight Java Game Library. Neid vaatame kahes järgnevas alpeatükis.

2.1 OpenGL

OpenGL on 2D ja 3D vektorgraafika visualiseerimise rakendusliides. Liides koosneb funktsioonidest, mis võimaldavad programeerijal defineerida kahe ja kolmemõõtmelisi geomeetrilisi objekte ning kontrollida kuidas need objektid hiljem kaadri puhvrisse (ingl. *framebuffer*) renderdatakse.

Selleks, et OpenGL abil ekraanile kaadrit renderdada tuleb kõigepealt luua graafikakontekst ning programmiaken. Need ülesanded jäävada aga OpenGL liidese vastutusalast väljapoole ja on operatsioonisüsteemi spetsiifilised. Õnneks on loodud mitmeid teeke, mis suudavad automaatselt nende ülesannetega hakkama saada. Selle jaoks kasutatakse teeki nimega GLFW läbi Lightweight Java Game Libary[3].

2.2 Lightweight Java Game Library

Lightweight Java Game Library (edaspidi LWJGL) on teek, mis annab Java programmeerijatele ligipääsu enimkasutatavatele graafika- ja multimeedialiidestele nagu OpenGL, Vulkan, OpenCL ja OpenAL. Lisaks sellele võimaldab LWJGL kasutada ka GLFW teeki, et tekitada graafikakontekst ning programmiaken. GLFW ja seega ka

---

[2] http://www.cs.cornell.edu/courses/cs312/2006fa/lectures/lec19.html

[3] https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf

LWJGL kaudu saab lugeda sisendit hiirelt, klaviatuurilt ja teistelt ühendatud kontrolleritelt[4].

LWJGL kasutamise vajaduse tingib asjaolu, et OpenGL funktsioonide kasutamiseks tuleb esmalt leida nende funktsioonide viide kasutades operatsioonisüsteemi spetsiifilisi funktsioone. Windows platvormil on selleks näiteks funktsioon `wglGetProcAddress`. See funktsioon võtab sõne kujul OpenGL funktsiooni nime ja tagastab viite funktsioonile kui selline eksisteerib. Java koodist ei saa me otse välja kutsuda `wglGetProcAddress` ning samuti pole Javas võimalik käsitleda viiteid. Küll aga võimaldab Java kasutada sellist liidest nagu Java Native Interface (JNI).

JNI on tarkvararaamistik, mis võimaldab Java virtuaalmasinas jooksval Java programmil kasutada operatsioonisüsteemi spetsiifilisi teeke[5]. JNI on sisuliselt see sild, mis võimaldab Java koodil ja OpenGL liidesel omavahel suhelda. LWJGL on lihtsalt üks abstraktsioon, mis peidab enda sisse ära OpenGL funktsioonide laadimise ja muu süsteemispetsiifika ning tekitab Java programmeerijale ilusa liidese, mida on mugav kasutada.

---

[4] https://github.com/LWJGL/lwjgl3-wiki/wiki

[5] https://www.baeldung.com/jni

3. Maastik andmestruktuurina

Rakenduse Voxel-World peamine funktsionaalsus on näiliselt lõpmatu ja muudetav maastik. Selle teostamiseks on vaja efektiivset moodust maastiku hoiustamiseks mälus, kasutajale visualiseerimiseks ning dünaamiliseks genereerimiseks. Käesolevas peatükis tutvustatakse maastiku hoiustamist ning visualiseerimist.

3.1 Maastiku tükid

Arvutil on lõplik mälu maht ja seega kogu maastikku korraga mällu laadida ei saa. See tähendab, et mängijale saab kuvada ainult temast kuni teatud kauguseni paiknevat maailma osa. Nähtavusraadiuse väärtus sõltub arvuti võimusest ja tuleb iga süsteemi jaoks sättida selliselt, et kasutaja näeks võimalikult suurt osa maailmast. Samas tuleb arvestades, et kaadrisagedus ei tohi langeda liiga madalale.

Mängija saab maailmas ringi liikuda ja see tähendab, et programm peab suutma töö käigus vaatevälja maastikku juurde laadida ning sellest välja jäävat maastikku mälust ära kustutada. Maailm koosneb kuupidest, mida nimetatakse ning seega on nähtavat osa maailmast mugav modeleerida massiivina, kus iga element kujutab ühte kuupi. Kuna isegi minimaalse nähtavusraadiuse korral jääb vaatevälja ligikaudu 300 000 kuupi, siis on vajalik nähtav maailm jagada väiksemateks osadeks. Neid tükke osi nimetatakse maastikutükkideks (ingl. *chunk*) ning eelnevalt mainitud minimaalseks nähtavusraadiuseks loen 7 maastikutüki küljepikkust. Sisuliselt oleks võimalik käsitelda nähtavat maailma ka ühe suure tükina, aga see oleks resursinõudlik, kuna iga mängja asukoha muutuse tõttu oleks vaja kogu massiiv ümber arvutada.

Iga maastikutükk kujutab endast $16 \times 256 \times 16$ (16 kuupi lai, 256 kuupi kõrge ja 16 kuupi pikk) kuubi suurust alla. Mis tähendab, iga maastikutükk vajab oma andmete hoidmiseks massiivi, mis mahutab 65536 elementi. Maastikutükid paikenvad kahemõõtmelises ruumis ning iga maastikutüki asukohta iseloomustab selle x- ja z-koordinaat. Kuna maastikutüki kõrgus on 256 ühikut, siis see tähendab, et maastik on küll lõpmatu horisontaalselt, aga mitte vertikaalselt. Kuigi maailm oleks võimalik teha lõpmatuks ka vertikaalsuunas, kui lisada maastikutükile y-koordinaat, mis näitaks maastikutüki kõrgust.

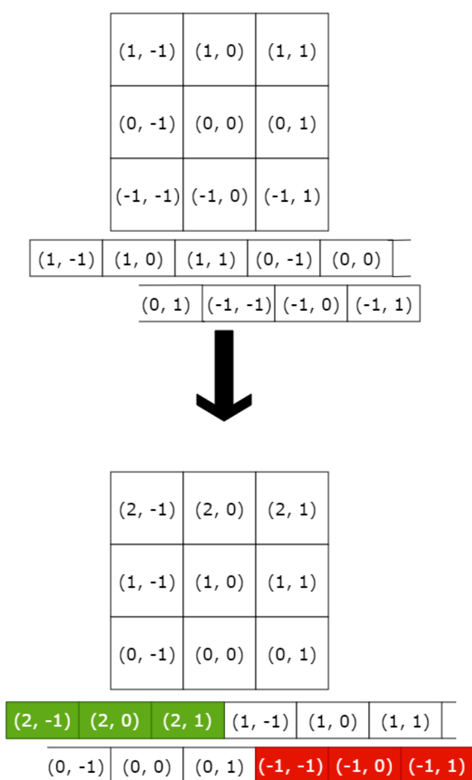
Maastikutükke laetakse ja kustutatakse iga kord, kui mängija liigub uue tüki peale. Maastikutükkide enda hoidmiseks on ka vaja sobivat andmestrutuuri. Kasutada saab

näiteks tavalist massiivi või paisktabelit. Mõlemal andmestruktuuril on oma eelised ja puudused.

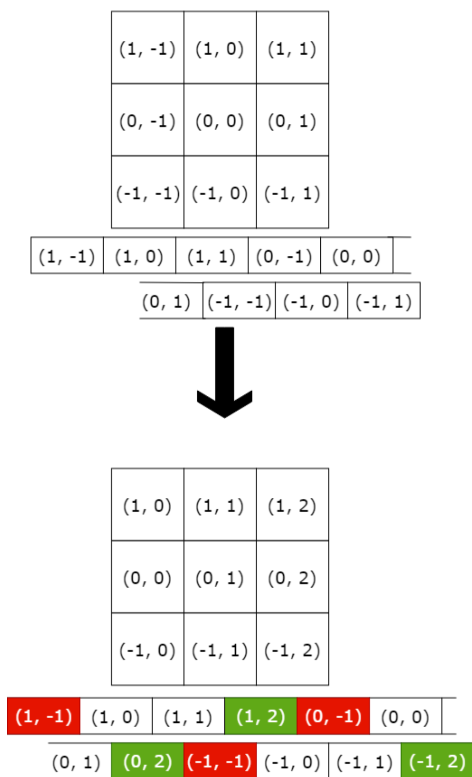
Maastikutükkide massiivis hoidmiseks on esmalt vaja nähtava maailma küljepikkust (ühikuks on maastikutükid). Selle kaudu saab arvutada massiivi suuruse. Näiteks, kui küljepikkuseks oleks 3, mis vastab implementatsioonis ühtlasi ka nähtavusraadiusele 1, siis maastikutükkide massiivi suurus oleks 9. Kusjuures maastikutükk, millel mängija paikneb oleks alati massiivi keskmine element. Mängija liikumisel naabruses asuvale maastikutükile tuleks nüüd kõiki elemente massiivis nihutada. Illustratsioon 1 näitab mängija liikumist maastiku z-teljel 1 tüki võrra positiivses suunas. Illustratsioon 2 kujutab massiivi elementide nihkumist, kui mängija liigub x-teljel ühe maastikutüki võrra positiivses suunas. Rohelisega on märgitud maastikutükid, mis juurde tulevad ning punasega märgitud tükid jäävad massiivist välja.

Sellise lähenemise puhul saab massivis elemente ümber tõsta Java standardteegi funktsiooniga `System.arraycopy` ning sõltuvalt mängija liikumissuunast saab lihtsa vaevaga välja arvutada juurde tulevate tükkide koordinaadid. Massiivis tuleb alati hoida ristküliku kujulist osa maailmast, kuigi suuremate raadiuste korral nurkades paikenvaid maasiktükke näha ei ole.

Maastikutükkide paisktabelis hoidmise eeliseks on see, et laadida saab täpselt need tükid, mida on konkreetselt vaja, et ekraanile pilti kuvada. Samuti pole paisktabelis vaja maastikutükke ümber tõsta, mis muudab koodi arusaadavamaks ja lihtsamaks. Samas, kui mängija asukoht muutub, siis on vaja kogu paisktabel läbi vaadata ning eemaldada need tükid, mis mängija nähtavusalasse enam ei mahu.



Illustratsioon 1 Massiivi elementide nihkumine liikudes mööda z-telge.



Illustratsioon 2. Massiivi elementide nihkumine liikudes mööda x-telge.

Voxel-World mängu implementatsioonis on kasutatud ka paisktabelit, kuna seda on lihtsam implementeerida ning koodi keerukust tasub tõsta ainult, siis kui see konkreetne koht põhjustab rakenduses jõudlusprobleeme.

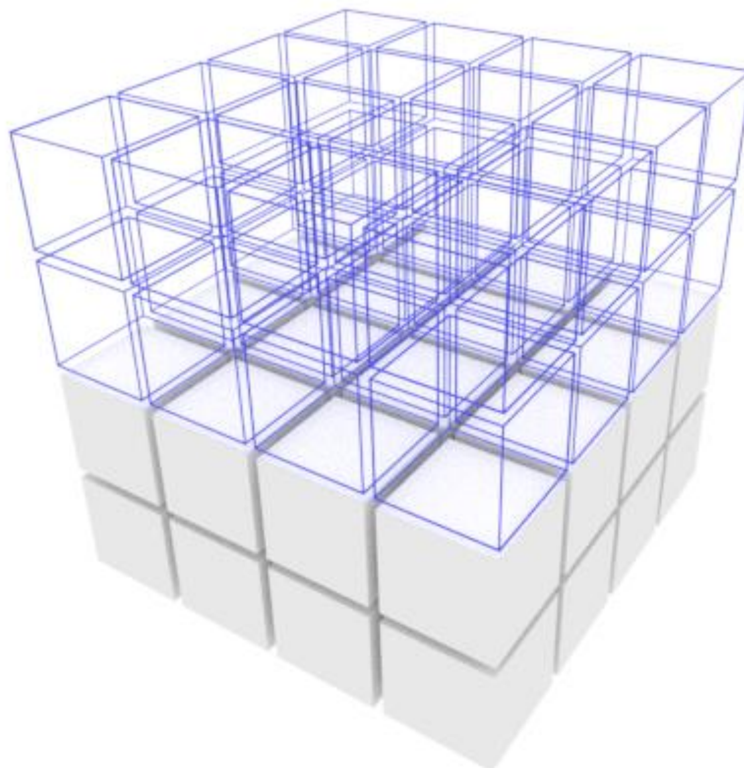
3.2 Võrestamine

Peatükk 3.1 selgitas kuupide ehk blokkide efektiivset hoidmist rakenduse siseseks kasutamiseks. Selles peatükis selgitatakse, kuidas käib blokkide massiivist kasutajale kuvatava pildi tekitamine.

OpenGL graafikakonveier võimaldab visualiseerida ainult geomeetrilisi primitiive, milleks on punkt, joon ja kolmurk [2]. Seega tuleb blokkide tüüpe sisaldav massiiv teisendada kolmnurkadeks, mida OpenGL abiga saab ekraanile kuvada. Seda protsessi kutsutakse

kolmnurkadest koosneva võrestiku genereerimiseks ehk võrestamiseks. Võrestik tekitatakse iga maastikutüki jaoks.

Renderdamise kiirus sõltub ekraanil olevate pikslite arvust ning visualiseeritavate primitiivide arvust. Kuna pikslite arvu saab vähendada ainult resolutsiooni vähendamisega, siis kõige parem viis renderdamist kiirendada on vähendada primitiivide arvu. Seega on hea kui võrestikud sisaldavad võimalikult vähe kolmnurki. Samas aga peab arvestama, et kui kasutaja eemaldab või lisab blokke, siis peab rakendus sellele muudatusele reageerima võimalikult kiiresti. Soovitavalt juba järgmiseks kaadriks. Kuna ainuke võimalus maastiku muudatusele reageerimiseks on võrestik ümber arvutada, siis peab võrestamine toimuma ka kiiresti [3].



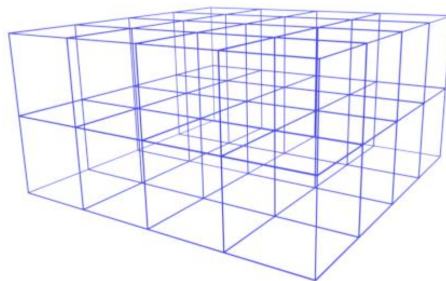
Illustratsioon 3. Hüpoteetiline maastiku mudel.

Järgnevalt selgitatakse kolme algoritmi, mida saab võrestiku genereerimiseks kasutada. Pildil Illustratsioon 3 on toodud näidiseks blokkide asetus maailmas. Blokid jagatakse kolme kategooriasse: õhk, maastik ning vesi. Iga kategooria renderdamine käib eraldi.

Pildil Illustratsioon 3 on traatmudeliga tähistatud õhk tüüpi blokid ning läbipaistmatute kuupidena on tähistatud maastik tüüpi blokid, mis peaksid mängijale maastikuna kuvatama. Kõik järgnevates alapeatükkides toodud võrestamise algoritmide näited tekitavad võrestiku pildil Illustratsioon 3 toodud maastiku mudeli jaoks. Samuti on illustratsioonidel tähistatud selguse mõttes ainult tahkusid mitte reaalselt programmis genereeritud kolmnurkasid. Tasub meeles pidada, et iga tahk koosneb kahest kolmnurgast.

Naiivne võrestamise algoritm

Naiivne võrestamise algoritm nagu nimest võib järeldada, pole eriti optimaalne. Probleemiks on see, et algoritm vaatab läbi kõik blokid maastikutükis ja genereerib kuus tahku kõikidele mitte õhk tüüpi blokkidele. See tähendab, et genereeritakse ka geomeetria nendele tahkudele, mis asuvad maastiku sees ning ei paista kuidagi kasutajale välja. Naiivse võrestamise tulemus on toodud pildil Illustratsioon 4. Algoritmi ainuke eelis on selle lihtsus [3].



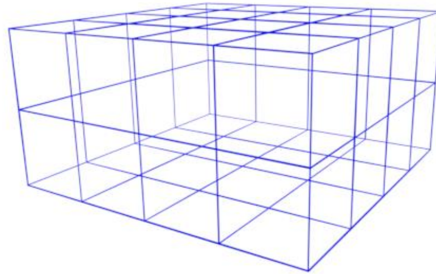
Illustratsioon 4. Naiivse võrestamise tulemus.

Naiivse võrestamise algoritmi poolt loodud võrestik on ebaefektiivne ning arvutimängu Voxel-World implementeerimisel kasutust ei leidnud. Siinkohal on naiivne algoritm välja toodud eelkõige järgnevates peatükkides selgitatud efektiivsemate algoritmidega võrdlemiseks.

Praakiv võrestamise algoritm

Praakiv algoritm erineb naiivsest selle poolest, et geomeetria genereeritakse ainult sinna, kus õhk tüüpi blokk puutub kokku mitte õhk tüüpi blokiga. See tähendab, et maastiku sisemised tahud, mis on peidus, jäävad võrestikust välja. Selle saavutamiseks tuleb iga mitte õhk tüüpi bloki puhul läbi vaadata kõik sellega tahkupidi kokku puutuvad naaberblokid. Eelmises alapeatükis naiivse meetodiga genereeritud võrestikul oli kokku 32 * 6 = 192 tahku, aga praakiva meetodiga genereeritud võrel on kokku 64 tahku –

kolmekordne erinevus isegi sellise väikse maastikutüki puhul. Muidugi, kui maastik on ebatasasem, siis ei pruugi võit olla alati eriti suur. Kõige halvemal juhul, kui maastik paikneb malelaua mustri sarnaselt, siis ei ole vahet, millist algoritmi kasutada. Tulemus on ikka sama. Õnneks selline halvim stsenaarium esineb praktikas väga harva. Enamasti on blokid maastikus tihedalt üksteise vastu pakitud [3].

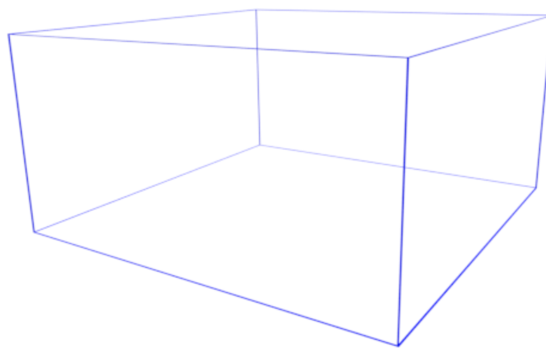


Illustratsioon 5. Praakiva võrestamise tulemus.

Praakivat võrestamise algoritmi on kasutatud ka Voxel-World-i implementeerimisel.

Agar võrestamise algoritm

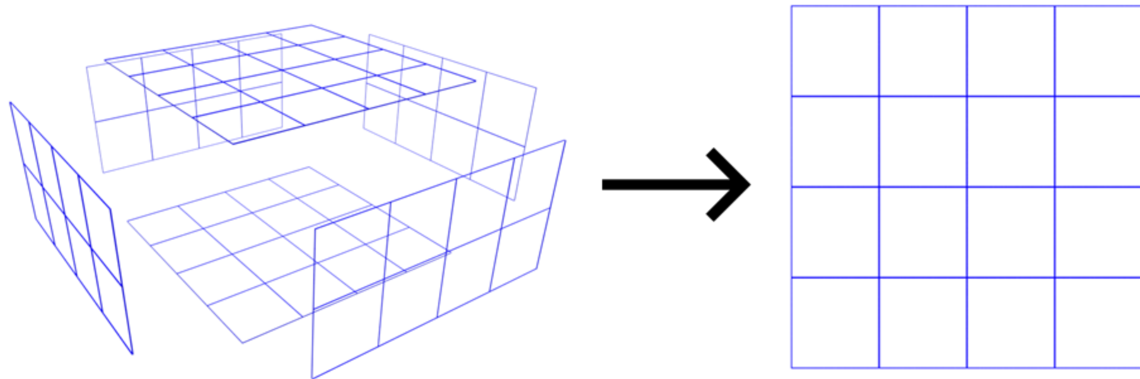
Agar võrestamine ühendab kõrvuti paiknevad sama tüüpi tahud kokku. Vähendades nii geomeetriliste primitiivide hulka. Pildil Illustratsioon 6 on näidatud ka agara võrestamise tulemus, mis antud juhul on ühtlasi ka optimaalne. Agar võrestamine ei pruugi alati anda optimaalset tulemust, aga selle algoritmi tulemus on alati vähemalt sama hea nagu praakiva võrestamisega [3].



Illustratsioon 6. Agara võrestamise tulmeus.

Agara võrestamise korral käiakse kogu maastikutükk igas suunas läbi ning tekitatakse praakiva meetodiga esialgne võre. Seejärel jagatakse tekkinud võrestik sektsioonideks, mis

taandab 3D probleemi 2D probleemiks, vt. Illustratsioon 7. Kõiki tekkinud 2D sektsioone käsitleme eraldi. Eesmärgiks on ühendada võimalikult palju tekkinud tahke kokku, et vähendada võres olevate tahkude arvu [3].

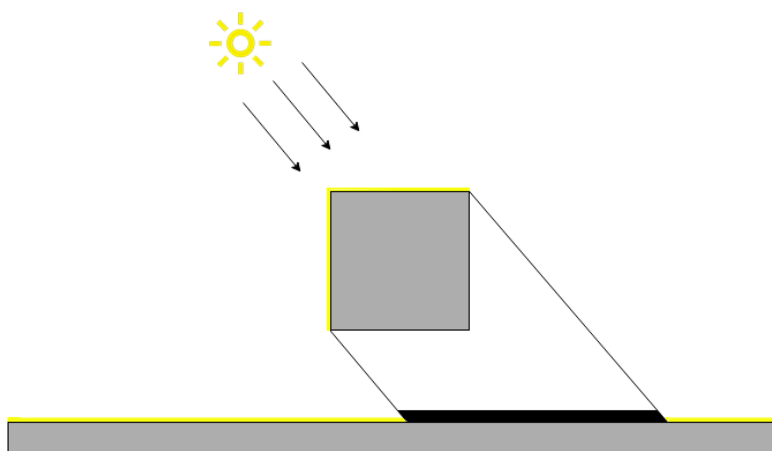


Illustratsioon 7. Agara võrestamise töö käigus võetakse võrest igal sammul ette ainult üks sektsioon.

Agar meetod tekitab kirjeldatud algoritmide hulgas kõige efektiivsema võre. Samas on selle algoritmi implementeerimine ühtlasi ka kõige keerulisem. Voxel-World mäng saavutas piisava kaadrisageduse ka praakiva meetodiga ning seetõttu jäi agara võrestamise kasutamine arendusest esialgu välja. Tulevikus parema jõudluse saavutamiseks on võrestamine üks koht, mida saab mängus täiustada.

4. Varjud

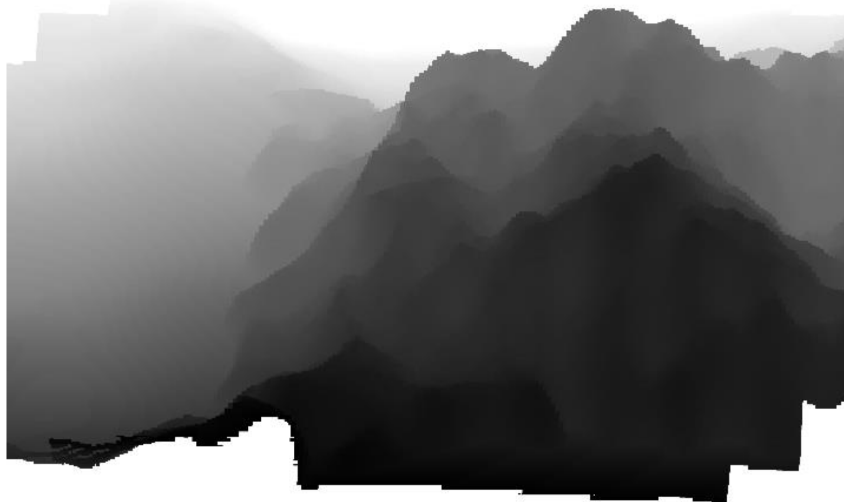
Voxel-World kasutab varjude näitamiseks varju kaardistamise (ingl. *shadow mapping*) meetodit. Varju kaardistamine on arvutigraafikas meetod varjude visualiseerimiseks reaalaja rakendustes[6]. See meetod võimaldab ligikaudset varjude visualiseerimist, mis on piisavalt hea mängude jaoks, kus füüsikaline täpsus pole eriti oluline [4].



Illustratsioon 8. Nooled näitavad langeva valguse suunda. Must värv kujutab tekkinud varju ning kollane toon näitab valgustatud pindasid.

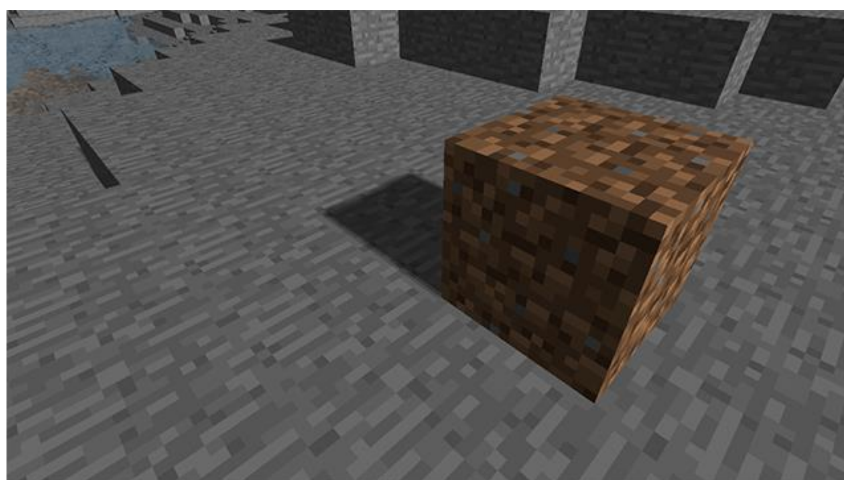
Varju kaardistamise meetodi idee seisneb selles, et stseen renderdatakse kahes etapis. Esimeses etapis on eesmärgiks leida iga stseenis nähtava punkti kaugus valgusallikast. Selle saavutamiseks renderdatakse kogu stseen läbi valgusallika asukohas paikneva kaamera tekstuurile, kasutades ainult fragmentide sügavusinfot. Tulemuseks on halltoonides tekstuur, mida nimetatakse sügavuskaardiks. Mida tumedam toon seda lähemal antud punkt valgusallikale asub (vt Illustratsioon 9). Oluline on seejuures fakt, et sügavuskaardile jäävad alles ainult valgusallikale kõige lähemal asuvad fragmendid. See tähendab, et valgusallika poolt on valgustatud ainult need fragmendid, mis asuvad sügavuskaardil [4].

[6] https://en.wikipedia.org/wiki/Shadow_mapping



Illustratsioon 9. Stseeni sügavuskaart.

Teises etapis renderdatakse kogu stseen nagu tavaliselt läbi mängija asukohas paikneva kaamera. Varjude tekitamiseks leitakse iga fragmendi kaugus valgusallikast ning seejärel võrreldakse seda kaugust eelmises etapis loodud sügavuskaardil oleva infoga. Kui sügavuskaardil samas asukohas paiknev fragment on lähemal, siis käesolev fragment on varjus. Vastasel juhul on antud fragment valgustatud. Tekkinud vari renderdatakse muust osast tumedamalt, vaata Illustratsioon 10.

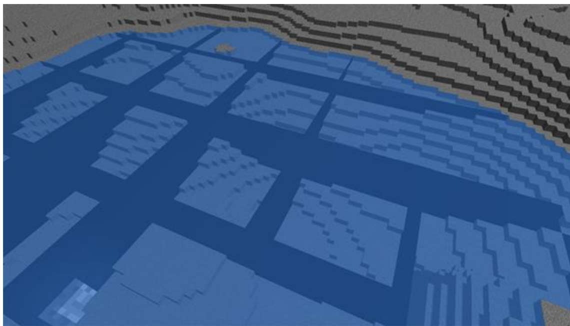


Illustratsioon 10. Kuubi vari.

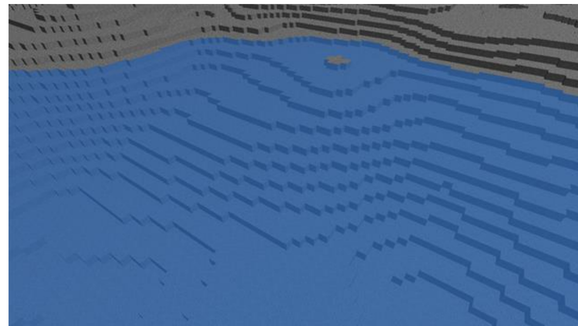
Varju kaardistamise meetodil on mitmed laialt levinud probleemid. Üheks selliseks probleemiks on varju akne. See väljendub triibulise mustrina nendes kohtades, mis ei ole varjus. Varju akne tekib kuna valgusallikas on pinna suhtes nurga all ning stseeni sügavuskaart on limiteeritud resolutsiooniga. Varju akne lahendamiseks tuleb sügavuskaardil rakendada väikest nihet, et tõsta seda natuke pinnast kõrgemale.

5. Vesi

Üks tehnika vee renderdamiseks on alfasujutamise kasutamine. Tuleb tekitada eraldi võrestikud vee ja maastiku jaoks ning seejärel aktiveerida sujutamine käsuga `glEnable(GL_BLEND)` ning seejärel rakendada õiget sujutamise funktsiooni. Antud juhul on selleks `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`. Sujutamise kasutamisel on oluline renderdada võrestikke õiges järjekorras. Õige järjekord on esmalt renderdada mitteläbipaistvad objektid nagu maastik ning seejärel läbipaistvad objektid nagu vesi. Vee ja maastiku võresid vaheldumisi renderdades on tulemus nagu pildil Illustratsioon 11. Kuna vesi on antud kohas kaamerale lähemal, kui maapind, siis sügavustestide käigus jäetakse alles ainult kaadripuhvris olnud vee fragmendid. Sujutamine siin ei tööta, kuna maapinna fragmentide alfa väärtused on 1.0. Jääb mulje nagu oleks maastikus auk. Õiges järjekorras renderdamise tulemus on näha pildil Illustratsioon 12.



Illustratsioon 11. Tulemus vale renderdamise järjekorraga.



Illustratsioon 12. Tulemus korrektse renderdamise järjekorraga

Realistliku vee visualiseerimise teevad eriliseks vee füüsikalised omadused nagu peegeldused ja refraktsioon. Samuti on vajalik, et vesi paistaks kasutajale pidevas liikumises, kuna suurtes veekogudes on täiesti tasast vett harva näha. Reaalajas vee visualiseerimiseks tuleb kasutada selliseid OpenGL funktsionaalsuseid nagu tekstuurile renderdamine ning pügamistasapinnad (ingl. *clipping plane*).

5.1 Refraktsioon ja peegeldus

Refraktsiooni ja peegelduste lisamiseks veele tuleb kogu steeni renderdada kolm korda. Esimesel kahel korral renderdatakse steen vastavalt refraktsiooni (Illustratsioon 14 B) ja

peegelduste (Illustratsioon 14 A) tekstuurile. Kolmandal läbimisel miksitakse need tekstuurid kokku (Illustratsioon 14 C) ja lisatakse vajadusel ka mingi toon (Illustratsioon 14 D).

Refraktsiooni tekstuuri jaoks on vaja renderdada ainult see osa stseenist, mis jääb vee piirist madalamale. Enne peegelduste tekstuuri renderdamist on vaja kaamera positsioon peegeldada vee tasapinna suhtes ning lisaks sellele pöörata kaamera tagurpidi, vaata Illustratsioon 13. Pärast kaamera õigele positsioonile asetamist tuleb tekstuurile renderdada kõik, mis jääb veepinnast kõrgemale.

Veepinnast kõrgemale ja madalamale jääva osa eraldamiseks tuleb kasutada pügamistasapinda. Tasapinna saab defineerida matemaatiliselt valemiga:

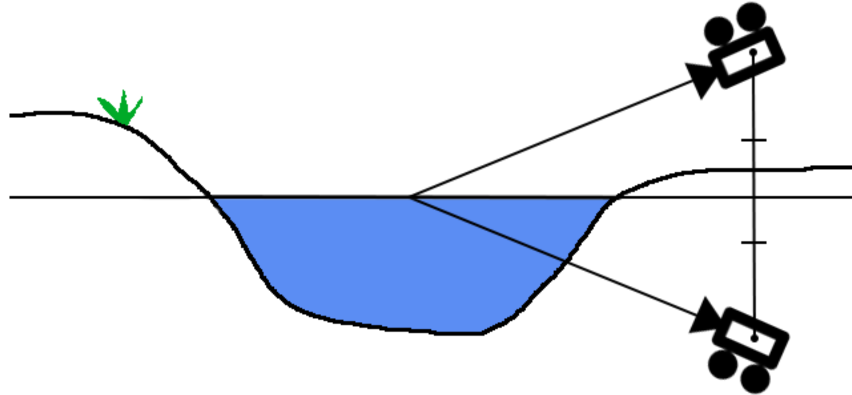
$$Ax + By + Cz + D = 0$$

kusjuures kolmik (A, B, C) määrab tasandi normaali. Kui tasandi normaal (A, B, C) on normaliseeritud, siis D näitab tasandi kaugust koordinaatide alguspunktist[7].

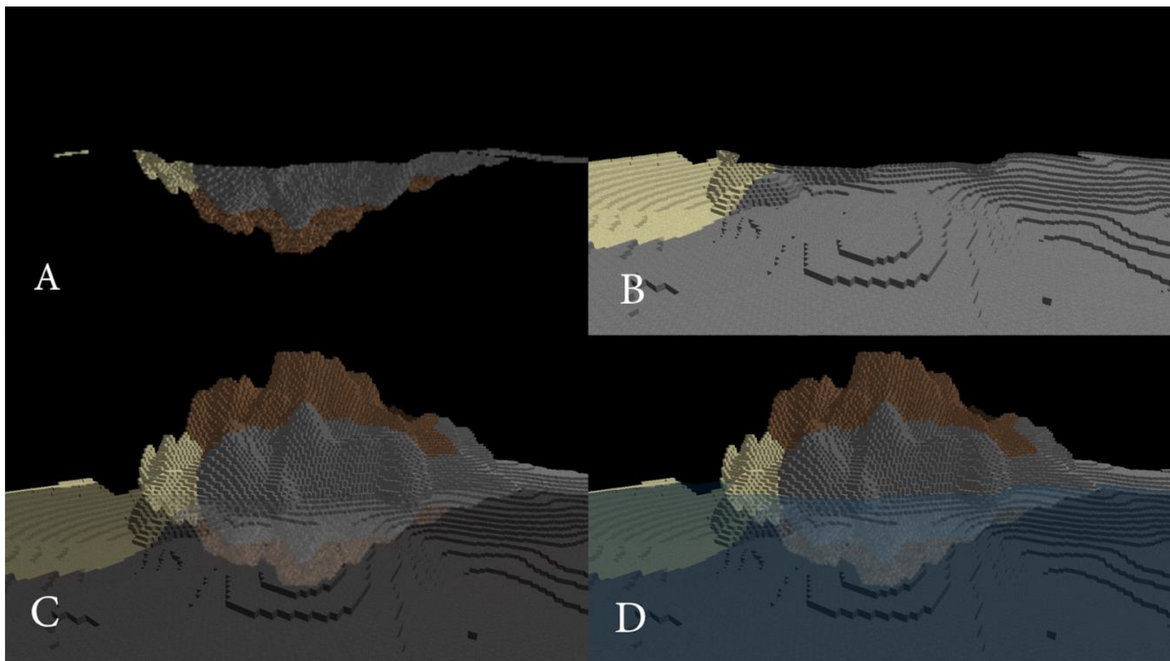
Geomeetria pügamine toimub varjutajas ja seega on vajalik saata sinna tasandi informatsioon. Selleks saab kasutada *uniform* muutujat. OpenGL varjutamiskeel (GLSL - OpenGL Shading Language) võimaldab pügamiseks kasutada `gl_ClipDistance` nimelist muutujat[8]. Varjutajas tuleb arvutada iga tipu asukoht tasapinna suhtes. Oluline pole kaugus vaid ainult see, et kas antud tipp on tasandinormaaliga samal pool tasapinda või mitte. Selle saavutamiseks piisab skalaarkorrutise leidmisest tipu asukohavektori ja tasandit iseloomustava vektori (A, B, C, D) vahel. Paneme tähele, et tasandit iseloomustab neljaliikmeline vektor aga tipu asukohta kolmemõõtmeline vektor. Selleks, et skalaarkorrutist leida tuleb tipust tekitada nelja elemendiga vektor, kus kolm esimest elementi on vastavalt tipu x, y ja z komponendid ning neljandaks liikmeks määrame alati 1.0.

[7] http://www.songho.ca/math/plane/plane.html

[8] http://docs.gl/sl4/gl_ClipDistance



Illustratsioon 13. Kaamera asukoht peegelduse ja refraktsiooni renderdamisel.

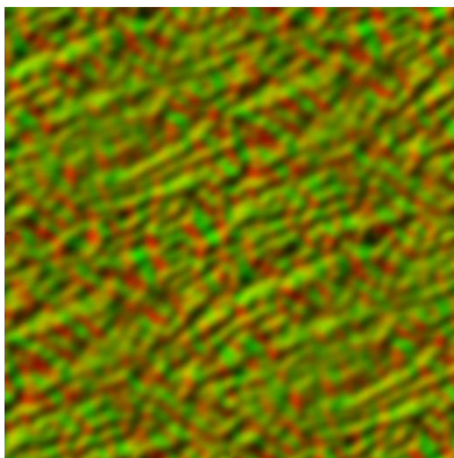


Illustratsioon 14. Vee renderdamise etapid: peegelduse tekstruur (A), refraktsiooni tekstuur (B), peegeldus ja refraktsioon kombineeritud muu stseeniga (C), veele lisatud sinine toon (D)

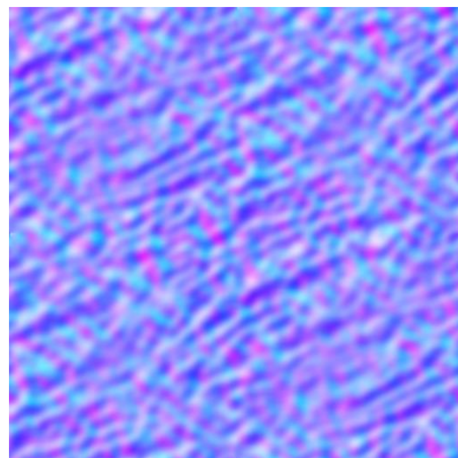
Tulemus on toodud pildil Illustratsioon 14 D. Sellel on olemas peegeldus ning vesi näeb välja läbipaistev, aga veekogu on peegelsile. See tundub natuke liiga ebaharilik, kuna looduses sellist tasast veepinda ei eksisteeri. Realistlikuma vee simuleerimiseks on vaja sellele lisada lainetus, mida vaadatakse järgmises alapeatükis.

5.2 Lainetus

Lainetust saab teha mitut moodi. Üks võimalus oleks reaalselt geomeetriat ja tippude asukohta ruumis muuta kasutades trigonomeetrilisi funktsioone ja ajas muutuvat sisendit [5]. Teine võimalus on lainetust simuleerida fragmendivarjutajas. Sellise lähenemise puhul tippude asukohta ei muudeta ning kogu efekt on ainult optiline. Lainetuse tekitamiseks rakendatakse refraktsiooni ja peegelduse tekstuuridelt sämplimisel väikest nihet, eesmärgiga moonutada visualiseeritavat pilti. Nihete jaoks kasutatakse varjutajas tekstuuri mida on näha pildil Illustratsioon 15. Selle tekstuuri punast ja rohelist komponenti kasutatakse sämplimisvektorite koordinaatide nihutamiseks.



Illustratsioon 15. Sämplimisvektori nihete tekstuur[9].

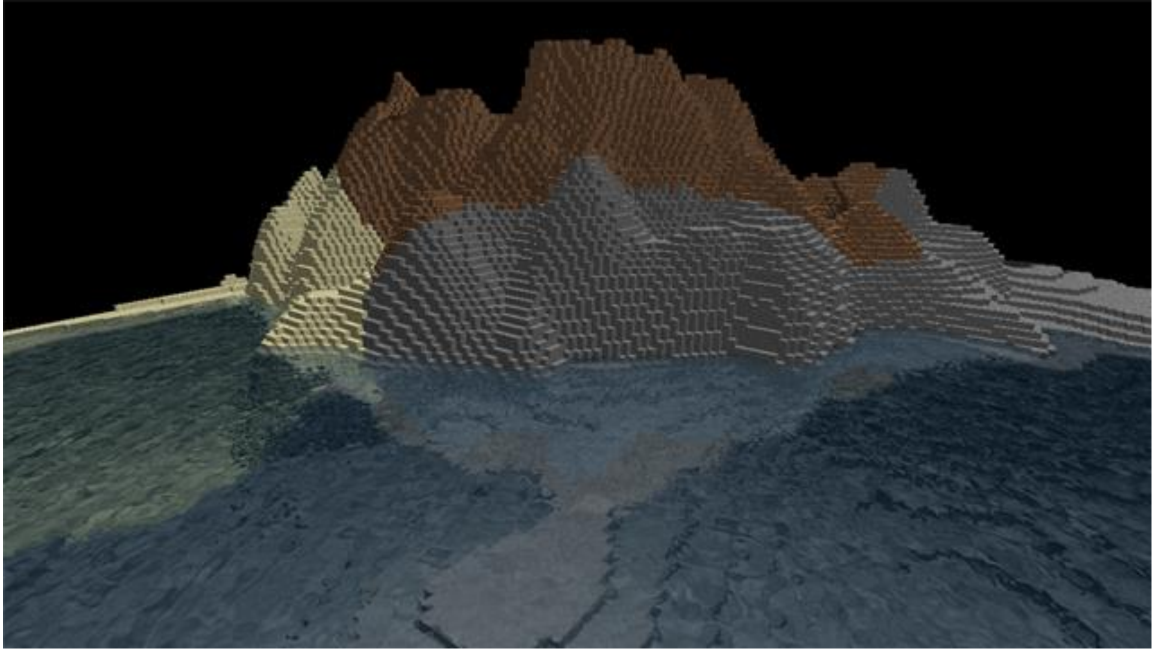


Illustratsioon 16. Normaalvektorite nihete tekstuur[10].

Selleks, et lainetus arvestaks valgusmudeliga on vajalik moonutada ka tasandinormaale. See tagab tulemuse, kus vee peal on visualiseeritud valgusallika sädelus. Tasandinormaalide moonutamiseks kasutab Voxel-World pildil Illustratsioon 16 kuvatud tekstuuri. Lõpptulemus on näha pildil Illustratsioon 17.

[9] https://www.dropbox.com/sh/lwvm5i223cwd5ue/AADedi_y3XTQ_j2aD2oH4DLKa?dl=0

[10] https://www.dropbox.com/sh/lwvm5i223cwd5ue/AADedi_y3XTQ_j2aD2oH4DLKa?dl=0



Illustratsioon 17. Vee renderdamise lõpptulemus.

Distantsilt vaadates on lõpptulemus piisavalt realistlik. Simuleeritud on kõik olulisemad vee omadused nagu peegeldus, läbipaistvus, lainetus ning ka valgusallika sädelus.

6. Kasutajaliides

Kasutajaliidese tähtsamad elemendid on nupud ja tekst. Kasutajaliidese kuvamine on muust steenist erinev kuna see koosneb enamasti kahemõõtmelistest ristkülikutest, millel on rakendatud sobivat tekstuuri. Samuti peab kasutajaliides olema alati kõige pealmine element.

6.1 Teksti renderdamine

Kuna OpenGL ei sisalda funktsionaalsust teksti renderdamise jaoks, siis teksti renderdamine OpenGL vahendusel on keerulisem, kui esmapilgul arvata võiks. Teksti renderdamise süsteem tuleb rakendusele eraldi juurde programmeerida. Voxel-World kasutab tekstuuril põhinevat fondi renderdamise süsteemi [6].

Tekstuuril põhineva fondi renderdamise jaoks on vaja eelnevalt koostada kaks faili: pilt, mis sisaldab kõiki mängus vajaminevaid tähti ning pilti kaardistav fail, mis seob vastavusse tähe koodi ja tähe parameetrid. Tähe olulisemad parameetrid on tähe asukoht pildi peal, tähe nihe rea joone suhtes ja kursori edasiliikumise ulatus (vt. Illustratsioon 19). Nende failide koostamiseks saab kasutada eraldiseisvat programmi näiteks BMFont[11].

Illustratsioon 18. Fondi tekstuur.

[11] http://www.angelcode.com/products/bmfont/

```
info face="Arial" size=32 outline=0
common lineHeight=32 base=26 scaleW=256 scaleH=256 pages=1 packed=0 alphaChnl=1 redChnl=0 greenChnl=0 blueChnl=0
page id=0 file="font_0.png"
chars count=103
char id=32   x=190   y=22    width=3     height=1     xoffset=-1    yoffset=31    xadvance=8     page=0  chnl=15
char id=33   x=9     y=90    width=4     height=20    xoffset=2     yoffset=6     xadvance=8     page=0  chnl=15
char id=34   x=127   y=105   width=10    height=7     xoffset=0     yoffset=6     xadvance=10    page=0  chnl=15
char id=35   x=108   y=47    width=16    height=20    xoffset=-1    yoffset=6     xadvance=15    page=0  chnl=15
char id=36   x=174   y=0     width=15    height=23    xoffset=0     yoffset=5     xadvance=15    page=0  chnl=15
char id=37   x=0     y=27    width=22    height=20    xoffset=1     yoffset=6     xadvance=24    page=0  chnl=15
char id=38   x=221   y=21    width=17    height=20    xoffset=1     yoffset=6     xadvance=18    page=0  chnl=15
char id=39   x=138   y=104   width=5     height=7     xoffset=0     yoffset=6     xadvance=5     page=0  chnl=15
char id=40   x=68    y=0     width=8     height=25    xoffset=1     yoffset=6     xadvance=9     page=0  chnl=15
char id=41   x=77    y=0     width=8     height=25    xoffset=1     yoffset=6     xadvance=9     page=0  chnl=15
char id=42   x=115   y=105   width=11    height=8     xoffset=0     yoffset=6     xadvance=11    page=0  chnl=15
char id=43   x=72    y=108   width=14    height=12    xoffset=1     yoffset=10    xadvance=16    page=0  chnl=15
char id=44   x=144   y=104   width=4     height=6     xoffset=2     yoffset=24    xadvance=8     page=0  chnl=15
char id=45   x=191   y=104   width=9     height=2     xoffset=0     yoffset=18    xadvance=9     page=0  chnl=15
char id=46   x=201   y=104   width=4     height=2     xoffset=2     yoffset=24    xadvance=8     page=0  chnl=15
char id=47   x=225   y=63    width=10    height=20    xoffset=-1    yoffset=6     xadvance=8     page=0  chnl=15
char id=48   x=187   y=67    width=13    height=20    xoffset=1     yoffset=6     xadvance=15    page=0  chnl=15
```

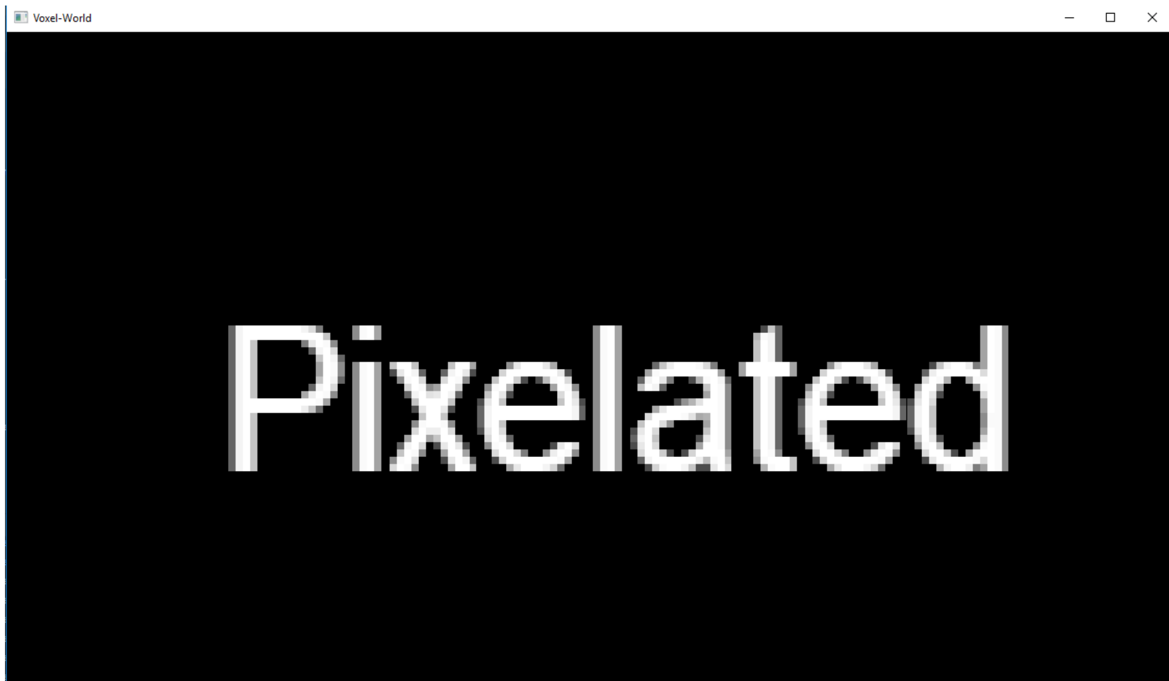
Illustratsioon 19. Fondi tekstuuri selgitav fail.

Renderdamiseks tuleb tekst jagada tähtedeks. Iga tähe jaoks tekitatakse seejärel sobiva laiuse ja kõrgusega ristkülik (vt.Illustratsioon 20). Ristküliku kõrguse ja laiuse jaoks võetakse andmed fonti kirjeldavast failist. Samuti tuleb arvestada, et kõik tähed ei paikne rea joone suhtes sama kõrgusel. Ristküliku tippudele pannakse õiged UV-koordinaadid vastavalt fondi pildile. Viimases etapis renderdatakse tähtede ristkülikud koos tekstuuriga ekraanile (vt Illustratsioon 21) [6]

Illustratsioon 20. Teksti piirjooned.

Illustratsioon 21. Teksti renderdamise lõpptulemus.

Kirjeldatud meetodi eeliseks on lihtne põhimõte, mis tähendab, et ka teostamine on lihtne. Puuduseks on see, et tekstuuris on kõik tähed eelnevalt kindlaks määratud resolutsiooniga. Suuruse muutmine toob kiiresti välja järsud üleminekud tähtede äärtes (vt.Illustratsioon 22).



Illustratsioon 22. Suured tähed liiga madalaresolutsioonilise tekstuuriga.

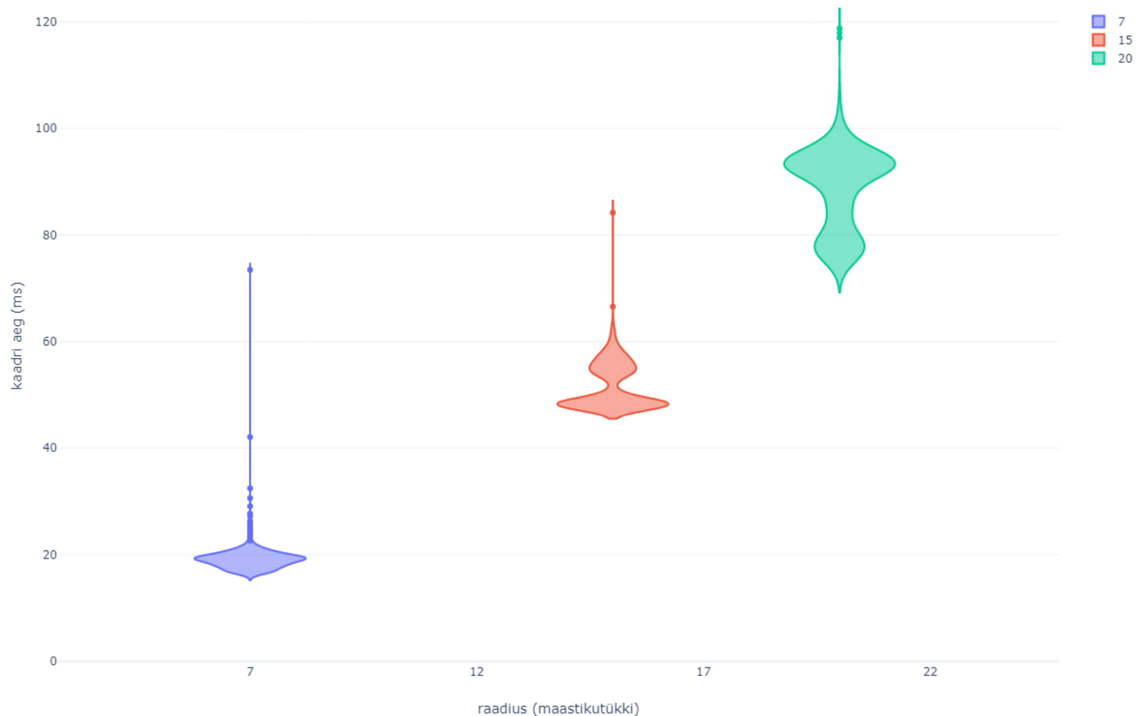
Järskude üleminekute vältimiseks tuleb fondi tekstuur tekitada piisava resolutsiooniga. Resolutsiooni valimiseks tuleb välja selgitada kõige suurem mängus vaja minev tähesuurus ning seejärel tekitada fondi tekstuurid.

6.2 Menüüd

Menüüd kujutavad arvutimängu erinevaid olekuid. Voxel-World Mängus eksisteerivad peamenüü ja pausi menüü. Pausi menüü võimaldab kiiresti mängu jätkata või siis hetkeks mängimist pooleli jätta. Peamenüü võimaldab mängimist alustada või mängu sulgeda. Menüüd koosnevad taustast ning nuppudest, mis võimaldavad liikuda mängu erinevate olekute vahel.

7. Jõudlus

Arvutimängude põhiliseks jõudluse mõõtmise vahendiks on kaadriaeg ning selle põhjal arvutatud kaadrisagedus [7]. Voxel-World puhul mõjutab kaadrisagedust peamiselt nähtava maailma raadius.



Illustratsioon 23. Mängu kaadriajad erinevate raadiuste korral.

Kaadriaja mõõtmiseks kasutatud arvuti tehnilised andmed on järgnevad:

- Protsessor: Intel Core i5-6200U
- Graafikakaart: Nvidia GeForce 940MX
- Mälu: 4,0 GB

Jõudluse testimisel kasutasin kolme erinevat nähtavusraadiust: 7, 15 ja 20 maastikutükki. Mängu resolutsioon kõigi kolme mõõtmise korral oli 1280 × 720 pikslit. Tulemused on näha graafikapildil Illustratsioon 23.

Raadiuse 7 korral jääb kaadriaeg suures osas alla 20 ms, mis tähendab, et kaadrisagedus on ligikaudu 50 kaadrit sekundis. Enamasti loetakse ideaalseks kaadrisageduseks 60 ja rohkem kaadrit sekundis[7]. See tähendab, et raadiuse 7 korral on mäng antud arvuti peal

küll mängitav, aga soovitav oleks nähtavusraadiust vähendada. Kui raadius tõsta 15 peale, siis kaadriaeg hüppab 50 ms ümbrusesse. See tähendab, et kaadrisagedus langeb ligikaudu 20 kaadrile sekundis. Roheline graafik, kus nähtava osa raadiuseks on 20 maastikutükki näitab, et sellise suure nähtavusraadiuse korral kaadriaeg kasvab veelgi, jäädes suures ulatuses vahemikku 70-110 ms. Kaadrisagedus on seega 9 - 14 kaadrit sekundis. Jõudlustestide põhjal võib järeldada, et antud arvutiga tasub nähtavusraadius hoida alla 7 maastikutüki.

Graafikapildil Illustratsioon 23 on näha, et enamus kaadriaegasid langeb kitsasse ajavahemikku, mida kujutab graafiku laienemine. Samas on ka näha, et mõned üksikud kaadriajad hüppavad väga kõrgele võrreldes graafiku laia osaga. Raadiuse 20 korral leidus kaadreid, mis võtsid aega rohkem kui 200 ms, aga need ei mahtunud graafikule ära. Selliste hüpete põhjuseks on Java automaatne prügikoristus. Antud probleem on laialt teada ning selle leevendamiseks oleks vaja Voxel-World mängus mälu kasutamist optimeerida. Selleks saab kasutada objektide basseini (ingl. *object pool*) programmeeri-mismustrit[12].

[12] https://www.reddit.com/r/programming/comments/1z6j37/on_garbage_collection_in_games/

8. Kokkuvõte

Käesoleva töö raames loodi arvutimäng Voxel-World, mis on inspireeritud maailmakuulsast arvutimängust Minecraft. Mängu põhiline funktsionaalsus on lõpmatu protseduuriliselt genereeritud maastik, mis koosneb blokkidest. Blokke hoitakse spetsiaalses andmestruktuuris ning selle andmestruktuuri põhjal luuakse võrestik. Võrestiku loomiseks uuriti kolme algoritmi, milleks olid: naiivne, praakiv ja agar. Voxel-World kasutab praakivat võrestamise algoritmi, kuna see on piisavalt lihtne ja efektiivne.

Voxel-World implementeerib ka varjude ja realistliku vee renderdamist. Varjude renderdamise jaoks kasutatakse varju kaardistamise meetodit. Vee visualiseerimiseks toodi välja kaks võimalikku lahendust. Esimene meetod kasutas alfasujutamist. Teine meetod kasutas peegelduste ja refraktsiooni tekstuure ning nihete tekstuure lainetuse jaoks.

Jõudluse testimisel selgus, et testimiseks kasutatud arvuti peal tasub nähtavusraadius hoida alla seitsme maastikutüki. Selle tulemusega ei saa rahule jääda kuna teises peatükis on mainitud, et nähtavusraadius 7 on minimaalne, mille korral mäng näeb veel adekvaatne välja.

Töö teema valikul lähtus autor oma huvist arvutigraafika vastu ning seetõttu on plaanis valminud mängu edasi arendada. Võimalikeks täiendusteks on efektiivsem võrestamine, efektiivsem mälu kasutamine ning parem varjude renderdamine. Samuti võiks lisada taimestiku genereerimise ning erinevad mitte mängitavad karakterid nagu loomad ja koletised.

Valminud arvutimänguga ei ole ma päris rahul, kuna lootsin, et jõuan rohkem funktsionaalsuseid implementeerida. Puudu jäi mänguseisu salvestamine ning võimalus seda hiljem taasavada. Samuti ei ole mängus hetkel seadistamise menüüd. Ka mängu jõudlus jääb kesiseks.

9. Viidatud kirjandus

[1] Sepp A. Protseduuriline lõpmatu maastiku genereerimine. Tartu ülikool, 2016.

[2] Segal M., Akeley K. The OpenGL Graphics System: A Specification. The Khronos Group Inc, 2019, pp 13.

[3] Lysenko M. 0 FPS. https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/ (22.03.2019)

[4] De Vries J. Learn OpenGL. https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping (01.05.2019)

[5] Effective Water Simulation from Physical Models. 2007. https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch01.html (12.03.2019)

[6] De Vries J. Lear OpenGL. https://learnopengl.com/In-Practice/Text-Rendering (01.05.2019)

[7] The Computer Graphics and Virtual Reality Lab https://cgvr.cs.ut.ee/wp/index.php/frame-rate-vs-frame-time/ (08.05.2019)

Lisad

I. Terminid

Alfasujutamine

protsess, mille käigus kombineeritakse läbipaistev esiplaanil olev värv kokku taustaga.

Fragment

pikslisuurune tükk rasteriseeritud primitiivist, mis võib jõuda ekraanile.

Geomeetriline primitiiv

kolmnurk, lõik või punkt.

Graafikakonveier (ingl. *graphics pipeline*)

kontseptuaalne mudel, mis kirjeldab kolmemõõtmelise objekti (kahemõõtmelisele) ekraanile visualiseerimist.

Võrestik

mingi objekti või eseme pinna kujutis kolmemõõtmelises ruumis, koosneb geomeetrilistest primitiividest.

Maastiku protseduuriline genereerimine

maastik luuakse täielikult arvuti abil programmi töö käigus.

Praakimine

tingimustele mittevastavate geomeetriliste primitiivide välja jätmine renderdatavast hulgast.

Pügamine

selektiivselt renderdamisoperatsioonide sisse/välja lülitamine huvi pakkuvas piirkonnas.

Rakendusliides

protokollide, funktsioonide ja tööriistade komplekt rakendustarkvara programeerimise lihtsutamiseks.

Tipuvarjutaja

etapp OpenGL graafikakonveieris, mis protsessib individuaalseid tippe.

Varjutaja

spetsiaalset tüüpi arvutiprogramm, mis määrab ära pildi valgustatuse ning värvid.

Võrestamine

võrestiku loomine.

II. Failid

Kaasapandud failid sisaldavad rakendust kaustas voxel-world. Rakenduse käivitamiseks liikuda kausta bin ning käivitada fail voxel-world.bat Windows platvormil või voxel-world Linux platvormil. Rakenduse käivitamiseks on vajalik eelnevalt installeerida Java 11 või uuem versioon.

Kaustas testimise-tulemused asub testimise graafiku koostamiseks kasutatud andmestik. Iga tulba päis kirjeldab nähtavusraadiust ning andmed näitavad mitu millisekundit kaader aega võttis.

Kaasa on pandud ka lähtekood, mis asub kaustas voxel-world-code.

III. Kasutusjuhend

W	Liigu edasi
A	Liigu vasakule
S	Liigu tagasi
D	Liigu paremale
Tühik	Liigu üles
Vasak Shift	Liigu alla
Vasak hiireklahv	Eemalda blokke
Parem hiireklahv	Lisa blokke
Esc	Kuva pausimenüü, kui hetkel mäng käib

NB! Blokke saab lisada ja eemaldada ainult kuni 5 bloki kauguseni mängija asukohast.

IV. Teadaolevad vead

- Varjude renderdamine ei tööta korrektselt
- Blokkide lisamine/eemaldamine on vahepeal ebatäpne osutatud suuna suhtes
- Maastikusse tekivad ringi liikudes juhuslikud tühimikud kuna maastikutükk kaob ära kuhugi
- Inteli integreeritud graafikakaartidel ei tööta vee renderdamine korrektselt

V. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Markus Saarniit,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose **Voxel-World – maailm kuubikutest**,

   mille juhendaja on Raimond-Hendrik Tunnel,

   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. 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 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

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

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

Markus Saarniit

10.05.2019