TARTU ÜLIKOOL
Arvutiteaduste instituut
Infotehnoloogia mitteinformaatikutele õppekava

Ahti Maa

# Graafika loomine ning optimeerimine arenduskeskkonnas Unity

**Magistritöö (15 EAP)**

Juhendaja: Margus Luik

## Graafika loomine ning optimeerimine arenduskeskkonnas Unity

**Lühikokkuvõte:**

Magistritöös antakse ülevaade linna ehitamise mängu graafika loomisest ning optimeerimisest, kasutades Unity arenduskeskkonda. Graafika loomisel rakendati Unity visuaalse programmeerimise vahendeid Shader Graph'i ja Visual effects Graph'i. Antud vahendite abiga loodi mängu kasutatavad materjalid, visuaalsed efektid kui ka graafika loomist lihtsustavad süsteemid.

Testiti loodud graafiliste lahenduste mõju kaadrisagedusele, kasutades erinevaid profileerimise vahendeid sh. Unity Profiler ning Nvidia Nsight Graphics. Testide tulemusi kasutati ka graafika optimeerimise juures. Töös anti ülevaade kogu optimeerimise protsessist, milles rakendati muuhulgas Unity ECS metoodikat ning HLOD süsteemi.

Võtmesõnad: Arvutigraafika, visuaalsed efektid, Unity

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

## Graphics creation and optimization in Unity development platform

**Abstract:**

This master's thesis provides an overview of the process of creating and optimizing graphics for a city-building game using the Unity Development Platform. The graphics were created using the Unity's visual programming tools Shader Graph and Visual Effects Graph. With the help of these tools, game materials, visual effects as well as graphics facilitating systems were created.

The frame rate of the created graphical solutions was tested using various profiling tools incl. Unity Profiler and Nvidia Nsight Graphics. The test results were also used for graphics optimization. The work provided an overview of the whole optimization process, which applied Unity ECS methodology and HLOD system.

Keywords: Computer Graphics, Visual Effects, Unity

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

## Sõnavara

C# Job System – C# tööjaotus süsteem. Antud süsteem lihtsustab koodi loomist, mis töötab mitmetel lõimedel korraga.

Burst Compiler – Kiirkompileerija. Võimaldab kompileerimise käigus koodi automaatselt optimeerida (Unity arenduskeskkonnas).

Entity component system (ECS) – entiteet komponent süsteem. Programmeerimise paradigma, mille kohaselt eraldatakse üksteisest süsteemid ning komponendid (andmed). Entiteetidega on seotud komponendid ehk andmed ning süsteemid töötlevad neid andmeid. Süsteemid töötlevad korraga enamasti kõiki entiteete, mis omavad kindlat komponentide kombinatsiooni.

Data Oriented Technology Stack (DOTS) – Andmetele orienteeritud tehnoloogiate kogum. Süsteem, mis hõlmab endas entiteet komponent süsteemi, C# tööjaotus süsteemi ning kiirkompileerijat.

Visual effects graph (VFX graph) – visuaalsete efektide graaf (VFX graaf). Unity arenduskeskkonnas olev tööriist, mis on suunatud visuaalsete efektide loomiseks, kasutades visuaalset programmeerimist.

Shader graph – varjundi graafik. Unity arenduskeskkonnas olev tööriist, mis võimaldab luua materjale ja graafilisi efekte, kasutades visuaalset programmeerimist.

Shader – varjutaja. Programm, mis tegeleb graafika kuvamisega.

Texture – tekstuur. Tekstuur on arvutigraafika kontekstis tekstuuri pikslite ehk tekslite massiv. Arvutigraafika loomise juures võib vaadelda seda kui digitaalkujutist, milles on salvestatud tekstuuri pikslitega seotud informatsioon (näiteks värvus - RGB).

Normal map – pinnanormaal kaart. Tekstuur, mis määratleb pinna normaali suuna.

Node – sõlmpunkt ehk graafi element. Sõlmpunktide ühendamisega luuakse programmi loogika, kui kasutatakse visuaalset programmeerimist.

Vertex – verteks ehk nurgapunkt. Graafika loomise kontekstis on verteks enamasti punkti asukoht, kus kaks või enam serva (või joont) liituvad. Juhul kui tegu on üksiku verteksiga ehk tema kõrval ei ole servasid, võib verteksit vaadelda kui punkti x,y,z teljel (3D objekti puhul).

Pixel – piksel ehk pildipunkt. Piksel on väikseim kahemõõtmeline osa pildist või kaadrist, mida on võimalik ekraanil kuvada.

Material - Materjal on tekstuuride ja muude objekti välimust kirjeldavate andmete kogum, mida kasutatakse reaalmaailmas olevate materjalide simuleerimiseks.

Overdraw – ülejoonistamine ehk ülekate. Osa graafika kuvamise ehk renderdamise protsessist, milles tuleb pikslit töödelda mitu korda (samal kaadril). Antud olukord tekib, kui objekt on näiteks läbipaistev ning tema taga asuvad teised objektid (mille töötlemine on samuti vajalik).

Point cloud – punktikogum. Kogum punktide koordinaate (3D objekti puhul x,y,z teljel).

Frame rate – kaadrisagedus. Kuvatavate kaadrite arv ajaühikus. Unity arenduskeskkonnas on mõõtühikuks kaadrit sekundis (frames per second - fps).

Frame time – kaadriaeg. Määratleb, kui kaua kulus arvutil aega ühe kaadri kuvamiseks. Unity arenduskeskkonnas on mõõtühikuks enamasti ms ehk millisekund.

Post-processing effects - järeltöötlus efektid. Antud efekte rakendadakse Unity's üldjuhul kaadri loomise viimases faasis, millal need muudavad kuvatava kaadri lõplikku välimust. Antud efektide abiga on võimalik graafikat oluliselt muuta, lisades näiteks peegeldusi või värvi korrektsioone.

Level of detail (LOD) – detailsuse aste. Määratleb objekti detailsuse vastavalt sellele, kui kaugelt seda vaadatakse. Enamasti on LOD süsteem seotud kaameraga ehk mida kaugemal on objekt kaamerast, seda madalam on objekti detailsuse aste.

Hierarchical level of detail (HLOD) – hierarhiline detailsuse aste. Antud süsteem võimaldab suure hulga väiksemaid objekte asendada (teatud kauguselt) ühe suure objektiga. HLOD võimaldab moodustada objektidest teatud hierarhia, mille tipus on näiteks üks vähe detailne objekt, mida kuvatakse ainult siis, kui see on kaamerast kaugel. Kui kaamera liigub objektile lähemale, laetakse antud objekt välja ning tema asemele laetakse sisse hulk väiksemaid (detailsemaid) objekte.

Frustum culling – kaamera vaateväljaga seotud optimeering. Antud optimeerimise tehnikat kasutades kuvatakse ainult objekte, mis asuvad kaamera vaateväljas.

Billboard – 2D pilt. 3D graafika optimeerimise kontekstis on tegu (enamasti nelinurkse) pildiga, mis on alati kaamera poole suunatud. Kasutatakse tihti viimase LOD astmena, kuna teatud kauguselt ei ole alati võimalik eristada 3D ja 2D objekti.

Imposter - pettepilt. Graafika optimeerimise kontekstis on pettepilt 3D objekti põhjal moodustatud lihtsustatud kujutis.

City-building game – linna ehitamise mäng. Mängu žanr, milles on mängija eesmärgiks asula (sh. linna) ehitamine.

## Sisukord:


1. Sissejuhatus 6
2. Töö eesmärgid ja nõuded 7
3. Ülevaade Unity arenduskeskkonnast 8
    3. 1. Ülevaade Unity andmetele orienteeritud tehnoloogiate kogumist (DOTS) 8
        3. 1. 1. Unity ECS 8
        3. 1. 2. C# Job System 10
        3. 1. 3. Burst Compiler 11
    3. 2. Objektorienteeritud programmeerimine Unity arenduskeskkonas 11
    3. 3. Ülevaade visuaalsest programeerimisest 12
    3. 4. Unity Shader Graph 13
    3. 5. Visual Effect Graph 14
4. Graafika loomine ja optimeerimine arenduskeskkonnas Unity 16
    4. 1. Ehitusmehhaanika kavandamine 16
    4. 2. Ehitusmehhaanika optimeerimine kasutades ECS lähenemist 17
        4. 2. 1. Ehitusmehhaanika testimine 18
    4. 3. ECS visuaalne programmeerimine mängu loomise juures 21
    4. 4. Tekstuuride loomine kasutades Substance Designer tarkvara 22
        4. 4. 1. Pinnase materjali tekstuuride loomine, kasutades Substance Designer tarkvara 23
        4. 4. 2. Ehitise materjali tekstuuride loomine, kasutades Substance Designer tarkvara 25
    4. 5. Unity Shader Graph materjali loomise juures 27
        4. 5. 1. Unity Shader Graph taimestiku materjali loomise juures 27
    4. 6. Unity Unity VFX graph mängu loomise juures 29
        4. 6. 1. Pilvesüsteemi loomine 29
        4. 6. 2. Objektide lisamise süsteemi loomine kasutades VFX graafi 33
    4. 7. Graafika optimeerimine 37
        4. 7. 1. LOD süsteemi rakendamine graafika optimeerimise juures 37
        4. 7. 2. HLOD süsteemi rakendamine programmi optimeerimise juures 37
    4. 7. 3. Ülekatte (overdrow) vähendamine 38
    4. 7. 4. Kogu mängu keskkonna testimine 41
5. Loodud graafiliste lahenduste tulevik 43
6. Kokkuvõte 44
Kasutatud allikmaterjalid: 45
    Lisa 1. Litsents 47

# 1. Sissejuhatus

Töö annab ülevaate linna ehitamise mängule mõeldud graafika loomisest ning graafika optimeerimisest. Töös selgitatakse välja need mängu elemendid, mis võtavad kõige enam ressurssi ehk jäävad takistuseks sujuva kaadrisageduse saavutamisel. Antud elemente püütakse optimeerida kasutades erinevaid graafika optimeerimise meetodeid ning Unity ECS lähenemist.

Kuna graafika töötlemisel leiavad kasutust nii protsessor (CPU) kui ka graafikaprotsessor (GPU), optimeeritakse töö käigus mõlema tööd, eesmärgiga tagada võimalikult kõrge kaadrisagedus. Sellest tulenevalt võetakse kasutusele ka Unity andmetele orienteeritud tehnoloogiate kogum (DOTS), mille osadeks on ECS, C# Job System ning Burst compiler. Töö annab ülevaate kõigist Unity DOTS komponentidest ning need võetakse ka mängu keskkonna loomise ja optimeerimise juures kasutusele. Töös antakse ülevaade ka optimeerimise tulemustest, mida mõõdetakse Unity profiler abiga, erineva võimsusega süsteemide lõikes (miinimum ja soovituslik).

Kuna DOTS mõjutab ennekõike protsessori kasutust, võetakse kasutusele ka graafika optimeerimisega seotud meetmed, et vähendada graafikaprotsessori koormust. Näiteks luuakse 3D objektidest erinevate detailsusastmega variatsioonid ning need võetakse kasutusele Unity detailsuse astme (LOD) süsteemis. Lisaks rakendatakse ka ECS'ile tuginevat HLOD ehk hierarhilist detailsuse süsteemi, mis annab võimaluse asendada mitmed väiksemad objektid teatud kauguselt ühe suurema objektiga. Kasutusele võetakse ka teisi graafika optimeerimise meetodeid, mille rakendamise vajalikkus selgitatakse välja mängu keskkonna testimise kaudu.

Ülevaade antakse ka graafiliste lahenduste loomisest, kasutades Unity visuaalsele programmeerimisele tuginevaid tööriistu. Nendeks on Shader Graph ja VFX Graph, mis võetakse antud töös graafika loomise juures kasutusele. Antud töövahendid kasutatakse materjalide, efektide, ning looduse loomiseks (näiteks pilved, taimestik).

Tarkvaraarenduse juures on tähtis nii tarkvara optimeeritus kui ka tarkvara loomise protsessi kiirus. Seetõttu püütakse ka optimeerida ennekõike kõige ressursi mahukamaid elemente. Samal põhjusel antakse ka ülevaade, kuidas on võimalik Unity poolt loodud vahendeid (näiteks Shader ja VFX graaf ning kaamera kontroller) enda mängu loomise juures rakendada sh. luua nende abil süsteeme, mis võivad mängu arendamist oluliselt kiirendada. Näiteks luuakse VFX graafi abil süsteem, millega on võimalik keskkonda suurel hulgal 3D objekte lisada.

## 2. Töö eesmärgid ja nõuded

Magistritöö eesmärk on luua ning optimeerida linna ehitamise mängu graafilised lahendused, mis töötaksid sujuvalt ka miinum-nõuetele vastava riistvaraga.

Tööle esitatavad nõuded on seotud mängu kontseptsiooniga ning enim levinud riistvaraga. Kuna mängu tulevaseks müügi platvormiks on Steam, tuginevad välja selgitatud miinimum ja soovitatavad nõuded Steam riistvara uuringule (Steam Hardware Survey).

**Riistvaraga seonduvad nõuded:**
1. Mängu keskmine kaadrisagedus peab olema vähemalt 30 kaadrit sekundis (kaadriaeg 33,3 ms), miinimum nõuetele vastaval riistvaral, resolutsiooni 720p (HD).
2. Mängu keskmine kaadrisagedus peab olema vähemalt 60 kaadrit sekundis (kaadriaeg 16,6 ms), soovitatavatele nõuetele vastaval riistvaral, kasutades resolutsiooni 1080p (Full HD).

| Miinimum-nõuded | Soovitatavad nõuded |
|---|---|
| Intel: Core i3 3240 | Intel: Core i7-960 |
| AMD: Phenom II X4 965 | AMD: FX 6350 |
| Nvidia: GeForce 750 ti 2GB | Nvidia: GeForce GTX 1060 |
| AMD: Radeon 7850 2GB | AMD: Radeon RX 480 |
| 4 GB RAM | 8 GB RAM |
| Windows 7 | Windows 10 |

Tabel 1. Loodava tarkvara miinimum ja soovitatavad nõuded riistvarale

**Mängu konseptsioonist tulenevad nõuded loodavale graafikale:**
1. Mängu kontseptsioonist tulenevalt võib korraga ekraanil olla ligikaudu 15000 ehitist. Mäng peab olema sujuv (keskmine kaadrisagedus peab olema vähemalt 30) kui ekraanil on 15000 ehitist.
2. Ehitatav ala peaks olema hinnanguliselt ligikaudu 100 km2.
3. Kaamera kaldenurka ei saa piirata kindlasse perspektiivi ehk mäng peab olema mängitav mängija silmadest kui ka väga kõrgest pealtvaates (kaamera võib olla hinnanguliselt ligikaudu 1 meetri kuni poole kilomeetri kõrgusel).
4. Mängitavas keskkonnas peab olema peale ehitiste veel vähemalt 50000 objekti, mis tagaksid keskkonna detailsuse ja mitmekesisuse.
5. Kõik Unity järeltöötlus efektid, mis täiustavad mängu graafikat, peavad olema sisse lülitatud.

Mängu põhimõtteks on linna ehitamine ning sellega kaasnevate ressursside haldamine. Sellest tulenevalt peab olema tagatud mängijale võimalus suurel hulgal ehitisi keskkonda lisada. Lisaks peab olema loodavas mängu keskkonnas piisavalt ruumi kuhu linn ja sellel olevad ehitised (sh infrastruktuur) rajada. Optimeerimis-meetodiks ei saa olla ehitiste arvukuse või ehitatava ala vähendamine ning vaatevälja piiramine, kuna sellisel juhul ei säiliks mängu algne kontseptsioon. Samuti ei tohiks optimeerimis meetodid vähendada (oluliselt) graafika kvaliteeti, mis tõttu ei saa olla optimeerimise meetodiks näiteks järeltöötlus efektide mittekasutamine või varjude eemaldamine.

# 3. Ülevaade Unity arenduskeskkonnast

Antud peatükk annab ülevaate Unity andmetele orienteeritud tehnoloogiate kogumist (DOTS), keskendudes ennekõike Unity ECS metoodika kirjeldamisele. Lühidalt käsitletakse ka objektorienteeritud lähenemist Unity arenduskeskkonnas. Ülevaade antakse ka visuaalsest programmeerimisest ning sellele tuginevatest rakendustest (Shader Graph ja VFX Graph).

## 3. 1. Ülevaade Unity andmetele orienteeritud tehnoloogiate kogumist (DOTS)

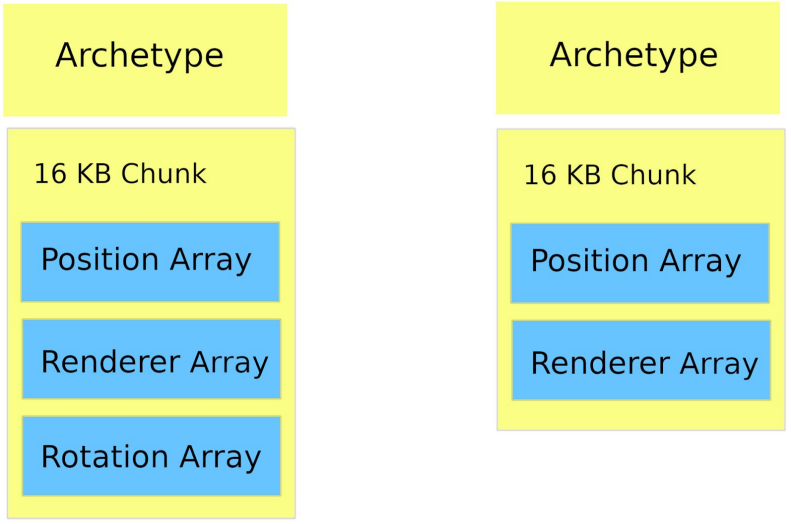
Unity andmetele orienteeritud tehnoloogiate hulka kuuluvad eniteet komponent süsteem ehk ECS, C# Job System ja Burst Compiler. Kõik kolm moodustavad kokku andmetele orienteeritud tehnoloogiate kogumi, mida tähistab akronüüm DOTS (Data Oriented Technology Stack). Kuna antud süsteem oli arendamisjärgus ning tuli sinna pakett (nimega Entities) lisada.

### 3. 1. 1. Unity ECS

ECS on üks peamiseid DOTS-i elemente, mida võib vaadelda kui programmeerimis paradigmat, mis erineb oluliselt Unity arenduskeskkonnas varasemalt kasutatavast objektorineteeritud programmeerimisest. Töö kirjutamise jooksul on ECS olnud arendusjärgus, kuid mitmed Unity arenduskeskkonna elemendid on juba kohandatud ECS-i lähenemise jaoks. Sinna alla kuuluvad näiteks objektide koordinaadistik ning selle muutmine, nende kuvamine ehk renderdamine ning ka Unity uus füüsika süsteem, mis on samuti kohandatud DOTS-i (sh. ECS-i jaoks). Siiski ei ole mitmed arenduskeskkonna osad veel sellega (töö koostamise ajal) liidetud – näiteks animatsioonid, mis tõttu tuleb praegu kasutada peale ECS metoodika ka objektorienteeritud lähenemist (Hybrid ECS).

ECS-i peamine erinevus objektorienteeritud programmeerimisest on andmete ning süsteemide eraldamine. Kui objektorienteeritud programmeerimises võivad muutujad ning meetodid olla mõlemad osad ühes samas klassis, siis ECS-i puhul on mõlema jaoks tehtud eraldi struktuur (struct). ECS-i lähenemise kohaselt kasutatakse ennekõike struktuure, et tagada efektiivne mälu kasutus (Ante, 2018). Entiteet komponent süsteemis on andmed ehk komponendid süsteemidest, kus kogu programmi loogika toimub, täiesti eraldatud. See võimaldab efektiivselt taaskasutada samu komponente (andmeid) erinevates süsteemides.

Komponente ehk andmeid liidetakse ECS-i lähenemises suurematekс andmekogumi tüüpideks (ingl *archetype*). Entiteet kuulub alati teatud andmekogumi tüüpi. Näiteks võib ühte entiteeti kirjeldada tema asukoha, välimuse ja rotatsiooniga (joonis 1). Need kolm komponenti (või komponentide massiivi) moodustavad andmekogumi tüübi. Süsteemid töötavad enamasti just andmekogumi tüüpidega, et saavutada võimalikult efektiivne andmete kasutus. Süsteemid võivad uuendada kõiki sarnaseid andmekogumi tüüpe korraga (Khalil, 2018). Andmekogumid asetsevad mälus lineaarsetel aadressidel (Khalil, 2018). Eemaldades mõne komponendi, muutub ka andmekogumi tüüp. Jooniselt 1 on näha, et näiteks pöörlemis-komponendi massiivi (Rotation Array) eemaldamisel, moodustub uus andmekogumi tüüp. Andmekogumi tüüp salvestatakse kindlas mälu tükis (ingl *chunk*), mis omab 16 kilobaiti (Khalil, 2018). Iga unikaale andmekogumi tüüp salvestatakse antud mälu tükis ning tänu andmete lineaarsele asukohale on neid võimalik ka vägagi efetiivselt töödelda (Khalil, 2018).




Joonis 1. Ülevaade andmekogumi tüüpidest.

Andmete muutumisel ei kontrollita reeglina üksikuid muutujaid vaid enne kontrollitakse, kas andmekogum on muutunud. Selleks muudetakse andmekogumi indeksit automaatselt, kui seda kogumit on muudetud. Kui kogum on muutunud, siis hakatakse vajadusel vastavaid toiminguid selles olevate andmetega tegema. See võimaldab eeldatavalt jõuda vajalike andmeteni kiiremini (Entities, 2019).

Entiteet on ECS-i lähenemises lihtsalt indeks (Khalil, 2018), mis seob endaga erinevaid andmeid. Erinevalt objektorienteeritud paradigmale tuginevast objektist ei ole entiteedil endal andmeid ega süsteeme. Entiteet on tänu sellele väga väikese andmemahuga võrreldes objektiga, mis omab reeglina Unity arenduskeskkonnas transformatsiooni komponenti (asukoht, kaldenurk).

Süsteemide abil luuakse mängu loogika. Süsteemid on komponentidest ehk andmetest eraldatud ning asetsevad üldjuhul eraldi C# struktuuris (ning eraldi failis). Juhul kui programmeerija ei määratle millised komponente (andmeid) või entiteete soovitakse töödelda, siis töödeldakse korraga kõiki entieete, mis kuuluvad teatud andmekogumisse. Näiteks kui tahetakse muuta entiteeti, mis omab asukoha ja rotatsiooni komponenti ning ei täpsustada täiendavalt otsingu parameetreid, siis töödeldakse kõiki sellesse andmekogumisse kuuluvaid entiteete. Kasutades näiteks meetodit "EntityQueryDesc" on võimalik otsingu tulemusi täpsustada (Entities, 2019).

Jagatud komponendid (ingl *Shared Components*) võimaldavad jagada entiteete vastavalt jagatud komponendi andmete väärtusele. Kõik entiteedid, millel on samad jagatud komponendi väärtused, on võimalik panna ühte mälu tükki. Näiteks võib olla 3D objektidel teatud hulk omadusi (materjal, varjud, kuju) tänu millele saab kõik antud entiteedid panna samasse mälu osasse. Antud põhimõtetele tugineb Unity Hybrid Renderer, milles kasutatakse jagatud komponete, et oleks võimalik sarnaste väärtustega komponente efektiivselt töödelda. Kui mõned jagatud komponendid omavad aga teistsuguseid (arvulisi) väärtusi, pannakse need eraldi mälu osasse ehk tükki. Sellest tulenevalt tõuseb mälu tükkide arv, mis tõttu ei tohiks jagatud komponente liigselt kasutada, kuna see võib muuta mälu kasutust ebaefektiivsemaks (Entities, 2019).

### 3. 1. 2. C# Job System

ECS on integeeritud ka C# tööjaotus süsteemiga, mis tagab turvalise lõimede kasutamise Unity arenduskeskkonnas. Lõimede võidujooks võib olla igasugune lõimede tegevus, mis teiste lõimede tööd negatiivselt mõjutab. Näiteks võib üks lõim muuta ning salvestada sama muutuja väärtust, mille kallal teine lõim töötas, mis võib kaasa tuua programmi ootamatu käitumise (Khalil, 2018). Selle automaatne vältimine vähendab oluliselt programmeerimiseks kuluvat aega ning tõstab ka programmi ohutust. Antud lõimede töö optimeerimine on integreeritud Unity ECS metoodikaga.

Unity ECS lahendus sisaldab mitmeid meetodeid, mis võimaldavad andmeid muuta, kasutades korraga mitmeid süsteemi lõimesid. Näiteks on ECS lähenemises kasutusel liides (ingl *interface*) “IJobForEach”, mida võib vaadelda kui järjendit, mis töötleb andmeid kasutades korraga mitmeid lõimesid. Lõimede korraga kasutamisel on aga teatud piirangud. Kuna entiteedile komponentide lisamise või eemaldamisega kaasnevad struktuursed muudatused, viiakse eniteet üle uude andmekogumisse. Struktuurseid muudatusi ei saa viia läbi kasutades töö lõimesid, kuna mõned lõimed võivad samu andmeid lugeda samal ajal kui teine lõim neid muudab. Sellest tulenevalt tuleb struktuursed muudatused läbi viia pea lõimel (ingl *main thread*), kasutades meetodit “EntityCommandBuffer”, mis viib muudatused ellu pea lõimel, siis kui töö lõimed on oma ülesanded lõpetanud (Entities, 2019).

### 3. 1. 3. Burst Compiler

ECS mustriga koodi kirjutamine võimaldab Unity poolt koodi kompileerida võimalikult efektiivsesse masinkoodi. Selleks on Unity integreerinud arenduskeskkonda lahenduse nimega Burst Compiler, mis töötab enamasti automaatselt, kui see arenduskeskkonnas aktiveerida. Ainuke nõue on lisada kiirkompileerimis käsk koodi: [UseBurstCopiler], peale mida püüab Burst Compiler koodi osa tõlgendada optimeeritud masinkoodi. Kasutades käsklust [BurstCompile] võib sellele anda ka parameetreid. Näiteks [BurstCopile(floatmode.fast)], mis võimaldab vähendada ujukoma arvude täpsust, et täiendavalt genereeritud koodi optimeerida.

## 3. 2. Objektorienteeritud programmeerimine Unity arenduskeskkonas

Objektorineteeritud programmeerimine on tihedalt integreeritud Unity arenduskeskkonnaga. C# on peamiseks programmeerimiskeeleks Unity arenduskeskkonnas, mille abil mängu loogika luuakse. Lisaks on masinõppega seotud toimingute programmeerimise juures kasutusel Python, mis kantakse samuti objektorienteeritud keelte alla. Programmeerimine enne ECS lähenemist on olnud objektorienteeritud. Kogu arenduskeskkonnas loodav programm (mäng) koosneb objektidest, millele lisatakse komponente. Objektid ise on enamasti klassid ning komponendid on samuti klassid, mis tihti pärivad ülemklassilt – Mono Behaviour (Unity manuaal, 2019).

Objektorienteeritud programmeerimise põhimõtted on tihedalt integreeritud ka Unity kasutajaliidesesse. Loodud objektile saab lisada komponente, milleks on klassid ning nende komponentide avalikud muutujad on enamasti kättesaadavad parameetrite näol Unity kasutajaliidesest. Antud lähenemine lubab kiiresti lisada objektile uusi komponente, mille avalikke muutujaid on võimalik reaalajas (programmi töötamise ajal) muuta.

Unity on loonud ka suurel hulgal erinevaid komponente (klasse), mida on võimalik mängu loomise juures kasutada. Näiteks on loodud muuhulgas heli, graafikaga, füüsika, rajaleidmise ja kasutajaliidesega seotud komponendid, mis võimaldavad antud süsteeme enda kirjutatud loogikaga kiiremini integreerida

## 3. 3. Ülevaade visuaalsest programeerimisest

Visuaalset programmeerimist on rakendatud erinevates, mängude loomiseks mõeldud arenduskeskkondades. Näiteks võib tuua Unity, Unreal Engine, CryEngine, Godot, Armory 3D ja Amazon Lumberyard, mis kõik annavad võimaluse visuaalset programmeerimist mingis programmi loomise faasis rakendada. Tihti annavad arenduskeskkonnad võimaluse luua programmi üldist loogikat ehk visuaalse lähenemise abil on võimalik luua näiteks terve mäng (Unreal Engine 4). Samuti on visuaalse programmeerimine aluseks erinevate visuaalsete efektide loomise juures (näiteks Unity VFX Graph).

Kuna visuaalne programmeerimine kujutab endas graafi loomist, on selle üheks tugevuseks ülevaate andmine programmi loogika liikumisest. Antud diagramm annab hea ülevaate kuidas (ajateljel) programmi töö liigub ning kuidas on erinevad süsteemid ja muutujad omavahel seotud.

Peamine viis kuidas programmi loogikat visuaalse programmeerimise abil luuakse on erinevate elementide ehk sõlmpunktide ühendamine omavahel. Reeglina on igal elemendil (node) sisendid millega element teeb mingisuguseid tegevusi. Antud tegevused võivad olla näiteks matemaatilised tehted või loogika rakendamine. Visuaalse programmeerimise elemendil on reeglina ka väljund, mis omakorda võib olla järgmise elemendi sisendiks. Omavahel ühendatud elementidest tekib voodiagrammi või protsessijoonist meenutav elementide kogum.

Üheks visuaalse programmeerimise miinuseks võib tuua väga suurte graafide rasket loetavust. Kui omavahel on ühendatud suur hulk elemente võivad elementide arvukus ning neid ühendavad jooned teha raskeks arusaamise elementide vahelistest seostest. Seetõttu on suurte graafide jagamine väiksemateks alamgraafideks soovituslik. Paljud visuaalse programmeerimise vahendid pakuvad võimaluse alamgraafide loomiseks ning nende kergeks taaskasutamiseks erinevates programmi osades (Substance Designer manuaal, 2019).

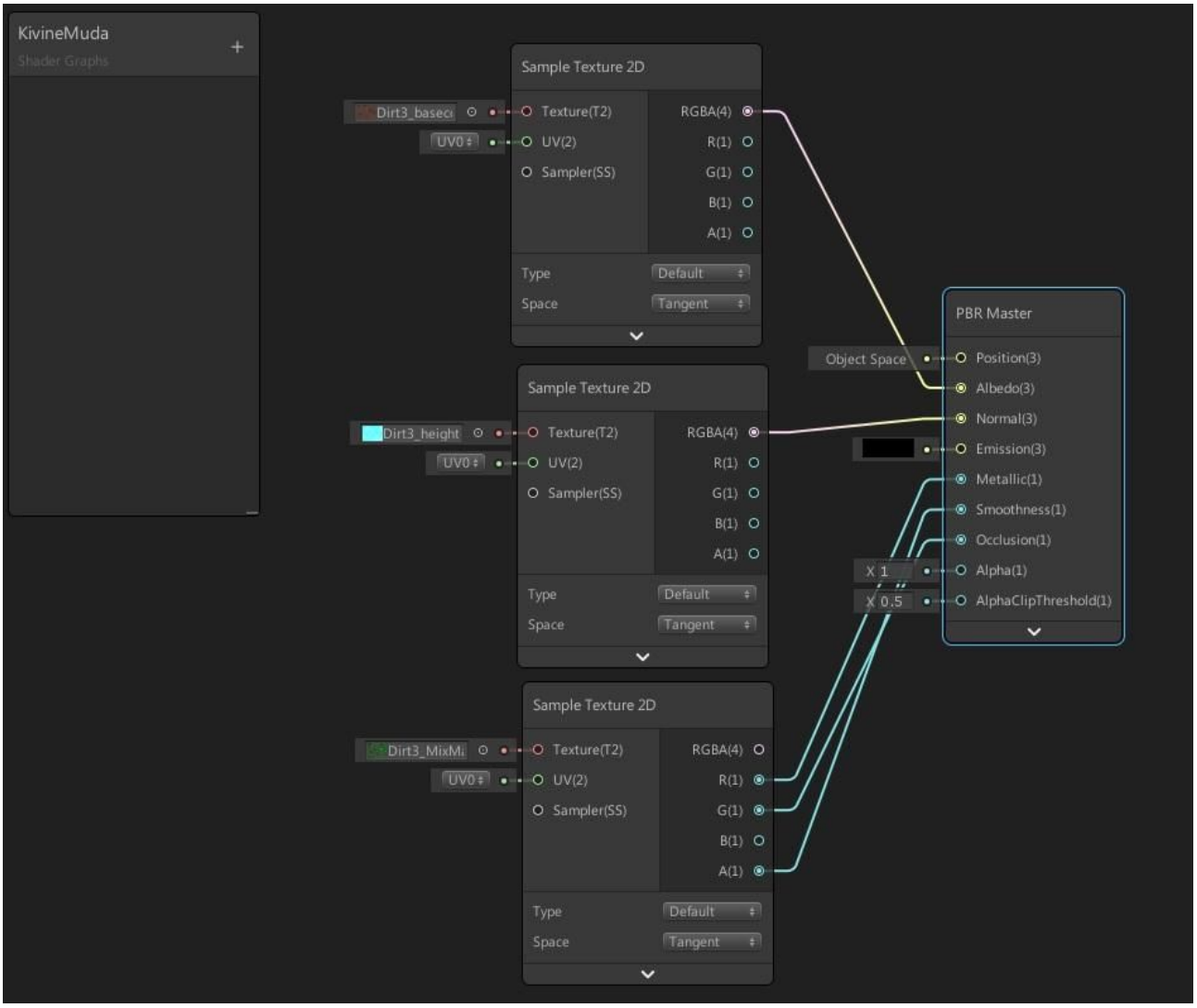
Visuaalne programmerimine leiab rakendust erinvates tarkvara loomise faasides sh. 3D modelleerimise juures kui ka materjalide ning tekstuuride loomisel. Näiteks kasutab visuaalset lähenemist materjalide loomise juures 3D modellerimis tarkvara Blender ning ainult materjalide loomisele keskenduv programm Substance Designer. Mõlemad on leidnud laialdast kasutust mängude loomise juures suurtes mängu stuudiotes ning andnud väga efektiivseid tulemusi. Antud tarkvara lahendustes on visuaalse programmeerimise eesmärk materjalide loomine võimalikult lihtsalt ja kiirelt, mis tõttu on ka visuaalse programmeerimise lahendused spetsialiseerunud neid eesmärke täitma. Näiteks võib üks element (ingl *node*) võtta koosinuse väärtusega sisendi ning selle abiga muuta värvuse parameetreid. Väljund võib olla sisendiks mõnele muule elemendile, mis teeb omakorda materjalis teatavaid muudatusi.

Substance Designer loob tekstuure kasutades selleks erinevaid protseduurilise müra (ingl *procedural noise*) funktsioone, näiteks Perlini müra (ingl *Perlin noise*). Samuti saab erinevaid müra funktsioone omavahel liita ja muuta nende väärtusi. Lisaks saab antud tarkvaras kasutada matemaatika- ja loogikafunktsioone materjalide loomisel (Substance Designer manuaal, 2019).

## 3. 4. Unity Shader Graph

Shader Graph'i abiga on võimalik luua arvutigraafikat, kasutades visuaalset programmeerimist. Visuaalne lähenemine asendab High Level Shader Language (HLSL) koodi kirjutamist. Seetõttu võimaldab antud töörist luua visuaalseid lahendusi ka inimestel, kes ei osaka HLSL keeles programmeerida. Antud graafikus liidetakse omavahel erinevaid elemente ehk sõlmpunkte (ingl *node*). Kõik elemendid liidetakse ühe väljund-elemendiga (master node), mis kujutab lõpliku materjali, mis on koostatud kõigist selle külge lisatud elementidest (Cooper, 2018).

Peamine ehk väljund-element (ingl *master node*), milleks võib olla PBR Master element (joonis 2), hõlmab endas kõiki füüsika seadustest lähtuvaid (ingl *physically based rendering*) materjali komponente ehk tekstuure, mida ka lõputöö käigus loodavas mängu keskkonnas rakendatakse. Iga materjali komponent annab edasi teatud informatsiooni, kuidas materjali tuleks kuvada. Peamisel elemendil on värvi ("Albedo"), varjundi ("Ambient occlusion"), pinna normaali ("Normal map"), sileduse ("Smoothness") ja metallilisuse ("Metallic") komponent. Kõigi komponentide liitmisel väljund elemendiga, valmib füüsiliselt korrektne materjal (Shader Graph manuaal, 2019).




Joonis 2. Näidis tekstuuride lisamisest väljund-elemendi (PBR Master) külge.

Iga tekstuur ei vaja kõiki värvi kanaleid (ingl *Red, Green, Blue, Alpha* - RGBA) ning võib sisaldada ka ainult ühe kanaliga seotud informatsiooni (näiteks R). Sellest tulenevalt võib pakkida kuni neli tekstuuri, mis omavad ühe kanaliga seotud informatsiooni, kokku üheks (RGBA) tekstuuriks. Tänu sellele on võimalik ühe tekstuuriga katta kuni neli kanalit (joonisel 1 on ühes tekstuuris rakendatud kolm kanalit ehk R, G ja A). Tekstuuride kokkupakkimine vähendab graafis olevate tekstuuride arvu.

Peamise elemendi külge liidetakse mitmed muud elemendid, mis võimaldavad muuta materjali omadusi. Paljudel elementidel on ka parameetrid, mida on võimalik teha ka avalikuks, mis annab võimaluse antud väärtusi Unity kasutajaliidesest muuta. Element võib viia läbi näiteks matemaatilisi tehteid (näiteks vektorite liitmine), hoida endas mingi objekti asukohta või muuta värvust (Lindman, 2018). Sõlmpunkte saab üksteisega liita ning neil on enamasti sisendid, mida element töötleb ning ka väljundid. Väljundid võivad tihti olla teiste elementide sisenditeks (Shader Graph manuaal, 2019).

## 3. 5. Visual Effect Graph

Visual Effect Graph ehk VFX graaf on olnud kasutatav Unity arenduskeskkonnas alates versioonist 2018.3 (Reilly, 2018). Antud lahenduse abil on võimalik luua oskaste süsteemidele tuginevaid visuaalseid efekte, kasutades HLSL koodi kirjutamise asemel visuaalset programmeerimist.

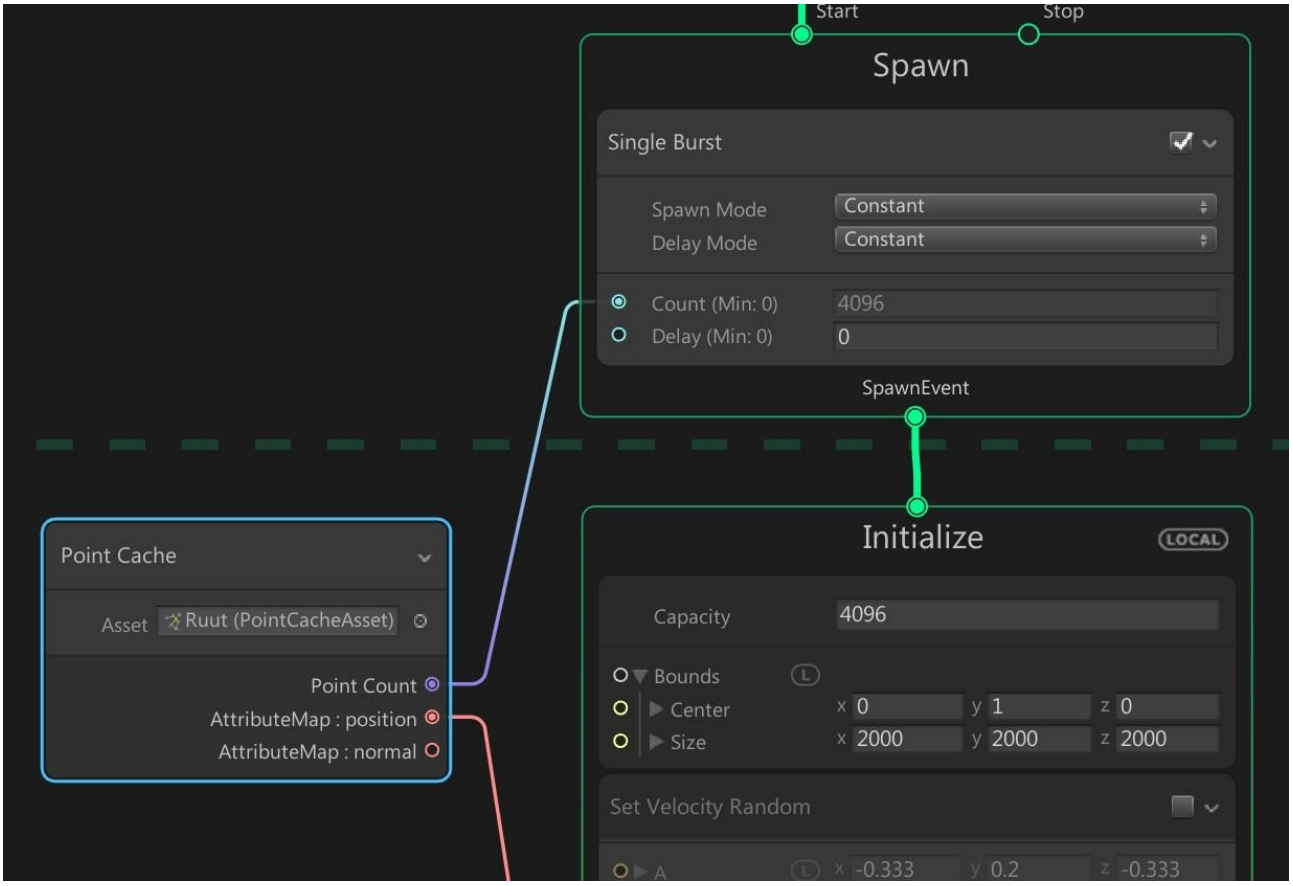
Antud töörist võimaldab aga visuaalse programmeerimise juurde liita ka HLSL koodi kirjutamise, läbi koodi elemendi (ingl *custom node*). HLSL koodi element on antud graafis nagu kõik muud elemendid ning ka sellel oma sisendid ja väljundid. Selles elemendis toimuv andmete töötlemine on määratletud HLSL koodi abil ning koodi on võimalik kergesti integreerida teiste graafi elementidega (Visual Effect Grap'i manuaal, 2019).

VFX graaf koosneb erinevatest osadest. Kõige tähtsam osa graafist on kontekstid (joonis 3), mis moodustavad süsteemi. Kontekstidele liidetakse tihti andme plokid, mis mõjutavad oluliselt kontekstide käitumist ning mis aktiveeritakse ülevalt alla. Operaatorid on kolmas osa graafist, mille ülesandeks on teha arvutusi, mille abil on võimalik plokkide tööd muuta (Visual Effect Graph'i manuaal, 2019).

Konteksti juurde lisatavaid andmeplokke on erinevat liiki. Kõige enam kasutatavad on atribuutide plokid. Loomise konteksti alla kuuluvad loomis plokid ("Spawn Block"), mis tegelevad uute objektide ekraanile kuvamisega. Kollisiooni konteksti kuuluvad kokkupõrke plokid, mis võimaldavad teha kindlaks, kas ühel visuaalsel osakesel on kokkupuude teisega ning kuidas peaksid need üksteisega reageerima. Lisaks on veel paljusid teisi andmeplokke, mis tegelevad oskeste suuruse, asukoha, rakendatava külgetõmbejõu, kiiruse ja elueaga (Visual Effect Graph manuaal, 2019).

Kolm konteksti – algatus (ingl Initialize), uuendamine (ingl Update) ning väljund (ingl Output) moodustavad koos süsteemi (Reilly, 2018). Antud süsteeme võib ühes graafis olla mitmeid. Algatus kontekst määrab ära, kui palju mälu osakestele eraldada ning kui suur hulk osakesi (ingl *particles*)

saab korraga süsteemis eksisteerida (Reilly, 2018).




Joonis 3. VFX graafi näide. Graafis on näha "Spawn" ja "Intitialize" konteksti, mis omavad "Single Burst" ja "Capacity" plokki. Mõlema kontekstiga on liidetud operaator "Point Cache", mis määrab osakeste arvu ja positsiooni.

Operaatorid tegelevad väärtuste arvutamisega andmeploki sees ning muudavad bloki toimimist. Operaatorid ühendatakse plokiga ning need asetsevad enamasti horisontaalselt. Operaatoreid võib omavahel ühendada ning neid saab muuta ka avalikeks atribuutideks, mida saab muuta programmi töötamise käigus, Unity kasutajaliidese kaudu. VFX graafis on kasutusel näiteks aritmeetika, geomeetria, trigonomeetria, loogika, värvi ja aja operaatoreid (Visual Effect Graph'i manuaal, 2019).

Graaf lubab määratleda, kuidas osakesed teiste osakestega kokku põrkavad. Selleks on olemas kollisiooni plokid (ingl *collision blocks*), milles saab määratleda, millise kuju põhjal kokkupõrget arvutatakse. Selleks võivad olla lihtsad kujundid näiteks kuup või silinder, kuid kollisiooni võib arvutada ka 3D tekstuuri (ingl *Signed Distance Field*) põhjal (Visual Effect Graph'i manuaal, 2019).

Graafis saab plokkide abil määratleda, millised füüsikalised jõud osakesi mõjutavad. Näiteks saab määratleda gravitatsiooni jõu sh. turbulentsi, mis genereeritakse müra valemite põhjal. Osakesi mõjutavaid jõude on võimalik arvutada ka vektorvälja (ingl *Vector Field*) põhjal, milles vektori suund määratleb ära osakesele kantava jõu suuna. Lisaks on võimalik rakendada osakestele ka külgetõmbe jõudu, mis on genereeritud 3D tekstuuri (ingl *Signed Distance Field*) põhjal (Visual

Effect Graph'i manuaal, 2019)

## 4. Graafika loomine ja optimeerimine arenduskeskkonnas Unity

Antud peatükis luuakse ning optimeeritakse linna ehitamise mängule mõeldud graafilised lahendused. See peatükk annab ülevaate ehitussüsteemi loomisest ning selle optimeerimisest. Lisaks antakse ülevaade mängu keskkonna tekstuuride loomisest, kasutades tarkvara Substance Designer ning materjali loomisest kasutades Unity Shader Graph rakendust. Kasutades Unity VFX graafi, luuakse nii pilvesüsteem kui ka vahend, mille abiga on võimalik kümneid tuhandeid objekte kiiresti keskkonda lisada. Lisaks antakse ülevaade mängu keskkonna optimeerimisest ning selle testimisest.

### 4. 1. Ehitusmehhaanika kavandamine

Antud peatükk annab ülevaate ehitussüsteemi loomisest. Näidete koostamisel on kasutust leidnud Unity API (ScriptReference, 2019). Mängus on kasutusel objektorienteeritud lähenemine, kuna kõik Unity osad ei ole veel liidetud ECS lähenemisega. Sinna alla kuuluvad näiteks animatsioonide loomine ning kasutajaliidese loomine.

Selle programmi (programmi näidiskood nr. 1) eesmärgiks on objektide lisamine keskkonda. Unity Scripting API on antud koodi kirjutamisel rakendust leidnud. Unity Scripting API omab mitmeid meetodeid, mille abil saab leida näiteks hiire positsiooni. Loodud klass "EhitisteLisamine", pärib peamiselt klassilt "MonoBehaviour". Edasi määratakse muutujale "hiirePos" väärtuseks hiire positsioon. Seejärel määratakse hiire positsioon ka "ray" väärtuseks. Muutuja ray on vektori algpunktiks ning "raycastHit" on vektori lõpp punkt, mis puudutab ristuvat objekti. Seejärel kontrollitakse, kas vektor on puudutanud ühtegi objekti (Collider) ning kas mängija on vajutanud hiire nuppu. Järgmisena luuakse objekt "instEhitis" samale asukohale, mida vektori lõpp punkt puudutas. Selleks, et mängijal oleks kergem objekte (ehitisi) luua kohakuti, ümardatakse instEhitis asukoha väärtus, kasutades meetodit "Round". Selle tulemusena paigutatakse objekt alati ühe mõõtühiku täpsusega. Joonisel 4 on näha, kui palju võtab kaadriaega 15000 ehitise kuvamine (antud tulemustest lähtudes, otsustati seda süsteemi optimeerida).

```
public class EhitisteLisamine : MonoBehaviour
{
    [SerializeField] private Camera kaamera;
    public GameObject instEhitis;
    void Update()
    {
        Vector2 hiirePos = Input.mousePosition;
        Ray ray = kaamera.ScreenPointToRay(hiirePos);
        //Ehitamine
        if (Physics.Raycast(ray, out RaycastHit raycastHit) && Input.GetMouseButtonDown(2))
         {
            instEhitis = Instantiate(ehitis, ray.GetPoint(raycastHit.distance),
Quaternion.identity) as GameObject;
            Vector3 realPosition = instEhitis.transform.position;
            // ümaradada ehitise asukoht ehk Grid Snapping
            instEhitis.transform.position =  new Vector3(Mathf.Round(realPosition.x),
Mathf.Round(realPosition.y), Mathf.Round(realPosition.z));
        }}}
```

Programmi näidiskood 1. Ehitise lisamine

## 4. 2. Ehitusmehhaanika optimeerimine kasutades ECS lähenemist

Alljärgnev koodilõik annab ülevaate koodist, mis võimaldab mängijal keskkonda ehitise luua. Optimeerimise eesmärgil on kasutusele võetud ECS ning keskkonda lisatakse nii "Game Object" kui ka entiteet. Objektilt on aga eemaldatud “Rendering” komponent, mille eesmärki (objekti visualiseerimist) täidab entiteet. Sellest tulenevalt näeb mängija entiteeti, aga mängu loogika võib olla liidetud objektiga.

ECS lähemise kohaselt tuleb olemasolevad objektid (Game Object) konverteerida entiteetideks. “GameObjectConversionUtility” meetod võimaldab stseenis olevad objektid muuta entiteetideks (koodi lõik 2).

Kogu koodi, mis on kirjutatud objektorienteeritud lähenemisega ei ole tarvis ümber kirjutada ECS metoodika jaoks, vaid valida kõige tähtsamad koodi osad, mille muutmine annaks suurima tulemuse (efektiivsuse tõstmise osas). Seetõttu ei konverteerita vektori loomist vaid konverteeritakse vektori asukohale loodav objekt entideediks. Antud süsteemis ei ole vektori loomine ressursi nõudev element, küll on seda aga loodav objekt, mis jääb programmi pikaks ajaks.

Tähtis osa antud koodist on ka entiteedi haldaja (Enity Manager) loomine Start meetodis. Antud element hoolitseb entiteetide töötlemise eest ning nende jagamiseks erinevatesse andme kogumitesse. Haldaja võimaldab hallatavatele entieetidele ning nendega seotud andmetele kiire ligipääsu, kuna entiteedid ja nendega seotud andmed grupeeritakse (Entities, 2019).

Suur osa koodist on sama, mis objektorienteeritud programmeerimist kasutatavas koodis, kuna selle eesmärk on sama. Tehtud muudatused ei mõjuta mängu ehitusmehhaanikat, küll aga on kood mõnevõrra efektiivsem, lähtuvalt testide tulemustest (vt. tabel 2).

Selleks, et liita ECS lähenemine olemasolevate objektorienteeritud süsteemidega võimalikult kiiresti, ei kohaldata kogu süsteemi ECS põhimõtetega. Näiteks on eelnevalt üles seatud PhysX Collider'id (Box Collider), millel on väga tähtis osa mängu toimimise juures (ennekõike liikluse süsteemi osas). Seetõttu ei konverteerita antud elemente ECS paradigmasse. Sellest tulenevalt ei pea tegema muudatusi teistes mängu süsteemides, mis on antud elementidega (Box Collider) seotud.

Koodi lõik 2. Alljärgnev näidiskoodi lõik annab ülevaate ECS ehitus süsteemist. Tegemist ei ole terve klassiga vaid antakse peamiselt Update ning Start meetodist.

```
public GameObject Ehitis;
public GameObject ECSehitis;
void Start()
{
var entityManager = World.Active.EntityManager;
}
private void Update()
{
float3 hiirePos = Input.mousePosition;
Ray ray = kaamera.ScreenPointToRay(hiirePos);

if (Physics.Raycast(ray, out RaycastHit raycastHit)  && Input.GetMouseButtonDown(2))
{
if (raycastHit.collider.tag == "EhitamiseksSobilik")
{
hiirePos = new float3(raycastHit.point.x, raycastHit.point.y, raycastHit.point.z);
//Entiteedi loomine
Entity entiteetEhitis = GameObjectConversionUtility.ConvertGameObjectHierarchy(ECSehitis,
```

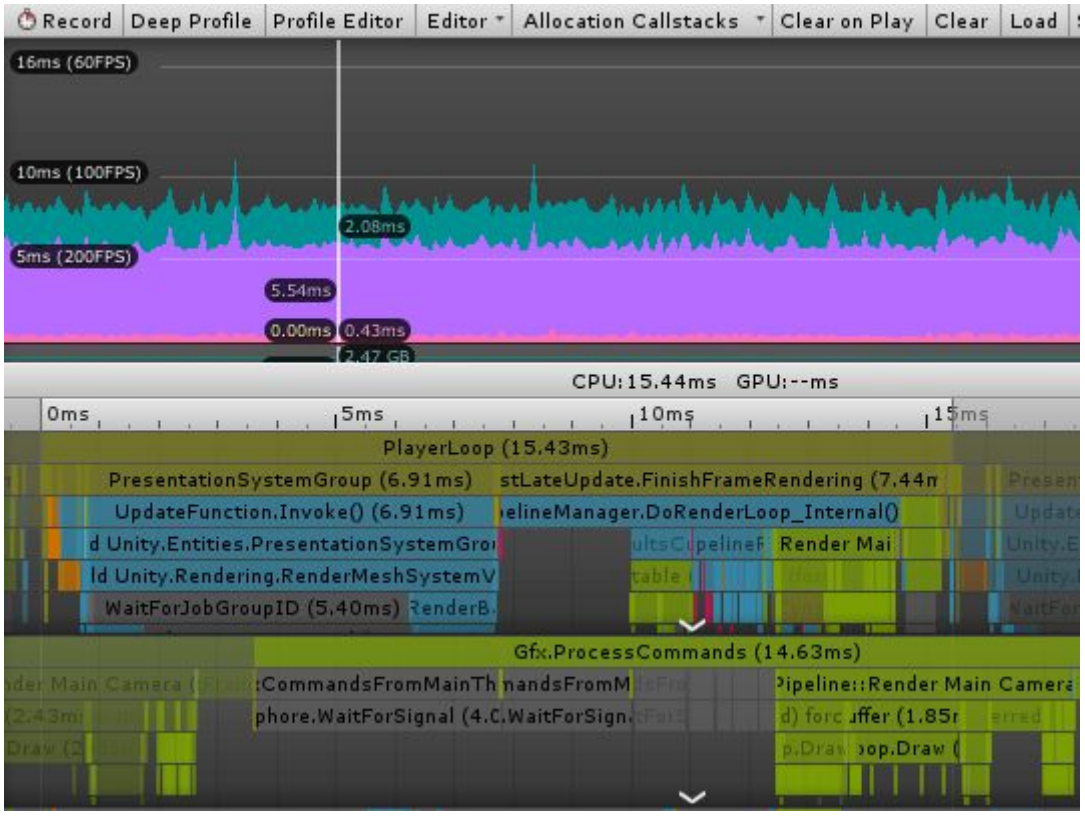
```
World.Active);
var entityManager = World.Active.EntityManager;
var instance = entityManager.Instantiate(entiteetEhitis);
var position = hiirePos;
var roundHiirpos = new float3(Mathf.Round(hiirePos.x), Mathf.Round(hiirePos.y),
Mathf.Round(hiirePos.z));
entityManager.SetComponentData(instance, new Translation { Value = roundHiirpos }); //kui ei
ole vaja ümardada, siis Value = position
//GameObjekt loomine - millelt on eemaldatud rendering komponent. See luuakse, et saaks
kasutada "Box Collider" komponenti ECS ehitussüsteemis.
GameObject objekt = Instantiate(Ehitis, ray.GetPoint(raycastHit.distance),
Quaternion.identity) as GameObject;
Vector3 realPosition = objekt.transform.position;
objekt.transform.position = new Vector3(Mathf.Round(realPosition.x),
Mathf.Round(realPosition.y), Mathf.Round(realPosition.z));
}}
else (UI.message("Sinna ei ole võimalik ehitada")) //Näitlik sõnum kasutajaliideselt
}
```

### 4. 2. 1. Ehitusmehhaanika testimine

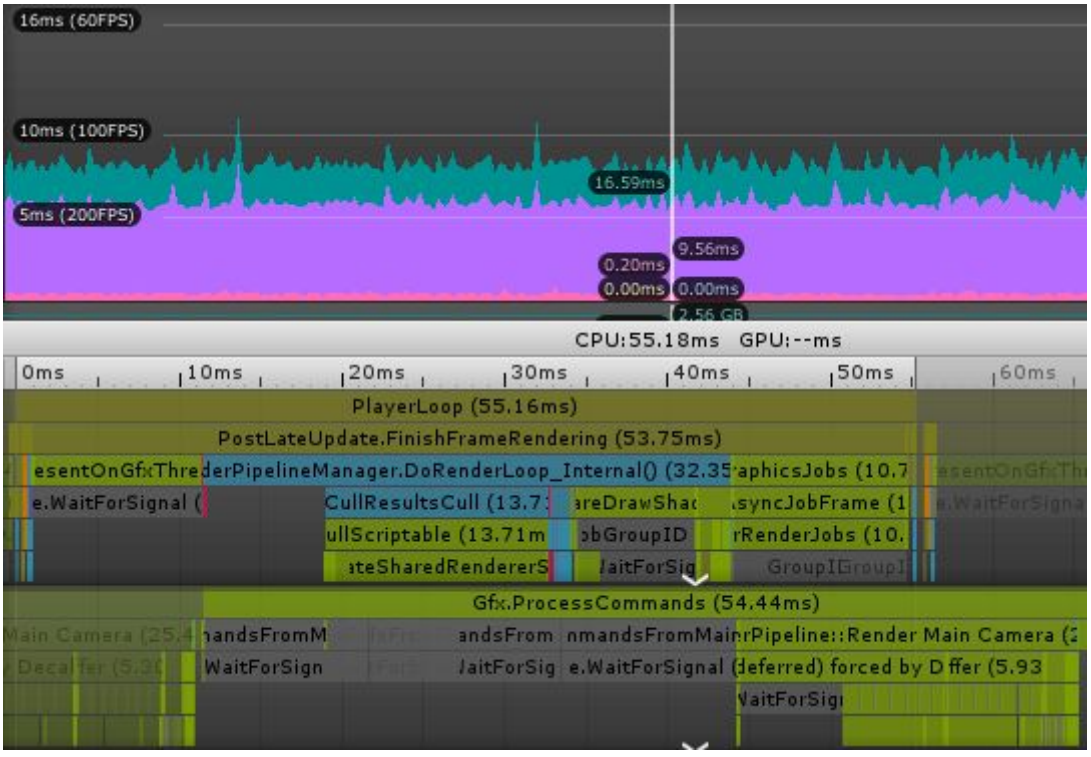
All-olevas võrdluses on näha, kui palju muutis ehitiste kuvamine kiiremaks DOTS-i kasutuselevõtuga. Ekraanil oli 15000 ehitist ning mägine maastik (Unity Terrain). Iga ehitisega on seotud "Box Collider" komponent ning 7500 ehitisele on lisatud Unity LOD süsteem ning ECS optimeeringus oli LOD süsteemid asendatud 7500 HLOD süsteemiga. All-olevas Unity Profile Analyzer laienduse vaates on võetud sisendiks nii ECS-iga optimeeritud (vasakpoolsed tulemused) kui ka optimeerimata programm (joonised 5 ja 6).

| Name | Left | < | > | Right |
|---|---|---|---|---|
| PostLateUpdate.FinishFrameRendering | 7.41 | | | 51.10 |
| PlayerLoop | 15.55 | | | 52.88 |
| RenderPipelineManager.DoRenderLoop_Internal( | 6.65 | | | 29.05 |
| Semaphore.WaitForSignal | 5.71 | | | 23.90 |
| EndGraphicsJobs | 0.54 | | | 13.12 |
| WaitForRenderJobs | 0.53 | | | 13.11 |
| Gfx.EndAsyncJobFrame | 0.53 | | | 13.11 |
| C#_CullResultsCull | 1.02 | | | 13.41 |
| CullScriptable | 1.01 | | | 13.41 |
| CullResults.CreateSharedRendererScene | 0.32 | | | 12.00 |
| WaitForJobGroupID | 6.59 | | | 17.78 |
| Gfx.WaitForPresentOnGfxThread | 0.00 | | | 9.08 |
| GC.Collect | - | | | 8.63 |
| UpdateFunction.Invoke() | 7.03 | | | 0.10 |
| PresentationSystemGroup | 6.88 | | | 0.04 |
| Default World Unity.Entities.PresentationSystem( | 6.87 | | | 0.04 |
| Default World Unity.Rendering.RenderMeshSyste | 6.73 | | | 0.03 |
| Shadows.PrepareDrawShadows | 0.05 | | | 4.16 |

Joonis 4. Unity Profile Analyzer vaade, mis on koostatud 298 mõõdetud kaadri põhjal (mõõtühik: millisekund). Vasakule poole on laetud ECS abil optimeeritud programmi testi tulemused ning paremale poole optimeerimata programmi tulemused. Testimiseks kasutatud süsteem: Intel i7 950, GTX 1060 6GB, 14 GB RAM.




Joonis 5. Unity profiler vaade, milles mõõdetakse ECS-i abil tehtud optimeerimist. Mõõdetud kaadri protsessori kaadriaeg oli 15.44 ms (Development Build)



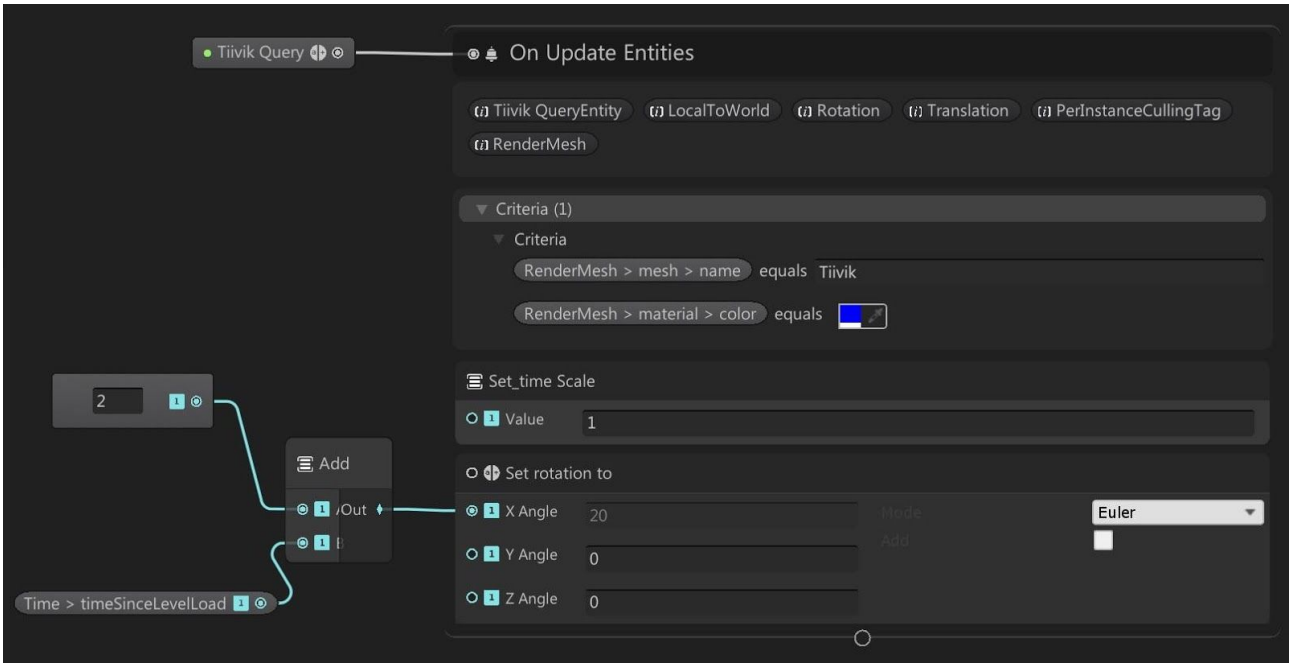

Joonis 6. Unity Profiler vaade, milles mõõdetakse programmi, mida ei ole ECS abil optimeeritud. Mõõdetud kaadri protsessori kaadriaeg: 55.18 ms (Development Build).

Pilt 1. Linna keskkond, mille kaadriaega mõõdeti.

Testidega selgitati välja optimeeritud ning optimeerimata ehitusmehhaanikaga seotud protsessori kaadriaeg. Optimeerimise tulemusena vähenes keskmine kaadriaeg 52,88 millisekundilt 15.55 millisekundile, mis oli eesmärke silmas pidades aktsepteeritav tulemus.

## 4. 3. ECS visuaalne programmeerimine mängu loomise juures

Unity on loomas ECS visuaalse programmeerimise lahendust, mis võimaldab kasutada DOTS-i eelised (Ans, 2019). Tegu on (töö koostamise käigus) veel eksperimentaalse lahendusega (Thierry, 2019). Antud lahendus aitab koostada mängu loogikat, erinevaid sõlmpunkte kokku liites. Lisaks on võimalik liidetud sõlmpunktidest genereerida ka ECS kood, mida saab hilisemalt muuta, nagu tavalist ECS koodi. Näiteks on võimalik vähem kui kümne noodi kokku liitmisel animeerida eniteete (näiteks tuulegeneraatori tiivik), mida demonstreerib järgnev graaf.




Joonis 7. ECS Visuaalne programmeerimine

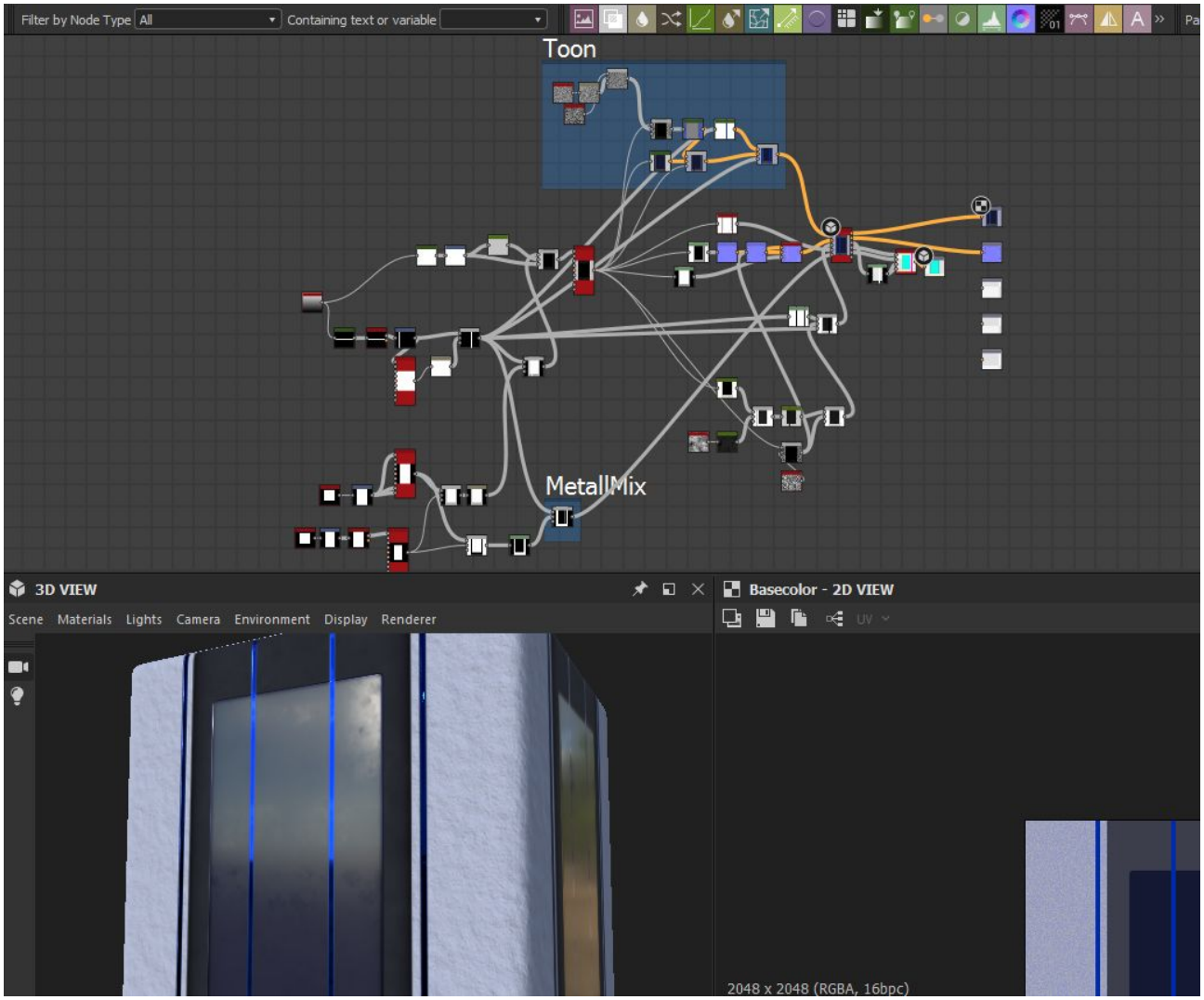
Antud ECS visuaalse programmeerimise graaf uuendab kõigi entiteetide asukohta, mis omavad teatud hulka komponente ning lisaks vastavad valitud kriteeriumitele. Graafi alguses määratletakse, milliseid komponente peaks entiteet omama, et selle asukohta graafi abil muutma hakatakse. Juhul kui graafis ei märgita täiendavaid kriteeriume (teises alalõigus), siis töödeldakse graafi abil kõiki (stseeni kuuluvaid) eniteete, mis omavad antud komponentide kogumit ehk, mis kuuluvad kindlasse andmekogumi tüüpi (ingl *archetype*).

Järgmises alalõigus on võimalik määratleda kriteeriumid, millele entiteet peab vastama, et seda saaks uuendada. Antud graafis on kriteeriumiteks nimi ning värvus, kuid neid võib olla tunduvalt rohkem. Kriteeriumite lisamine võib aga oluliselt suurendada protsessori töö mahtu, eriti kui kriteeriumiks seada näiteks entiteedi verteksite arv.

Järgmisena on graafile lisatud plokk “Set_time Scale”, mille eesmärgiks on määratleda aja kulgemise kiirus. Seejärel määratletakse järgmises graafi osas objekti pöörlemiskiirus (“Set rotation to”). Pöörlemiskiiruse seadmiseks on graafile lisatud kolm noodi. Omavahel liidetakse muutuja väärtusega 2 ning sõlmpunkt, mis määratleb aja möödumise väärtuse, alates stseeni laadimisest. Liitmise tulemus määratleb pöörlemise kiiruse. Kui aja noodi ei liideta, antakse entiteedile lihtsalt uus kaldenurga väärtus (antud juhul oleks objekti pööratud 20 kraadi x teljel).

## 4. 4. Tekstuuride loomine kasutades Substance Designer tarkvara

Kokku loodi lõputöös Substance Designer abiga 18 graafi. Valminud graafid saab jaotada kahte gruppi. Esimene neist on pinnase materjaliga seotud graafid ning neid loodi 7 tükki. Teise grupi moodustavad ehitistega seotud graafid, mida loodi 11. Mõlema grupi graafidest antakse selles peatükis ülevaade. Ehitiste materjalid koosnesid erinevatest osadest ning tekstuuridest. Näiteks võis ehitiste materjal (pilt 2.1) kujutada betooni, kiviseina, metalli ning võis omada aknaid ning muid elemente. Pinnase materjalideks olid lumi (2 graafi), kivid (2 graafi), liiv, savi ning mudane pinnas.

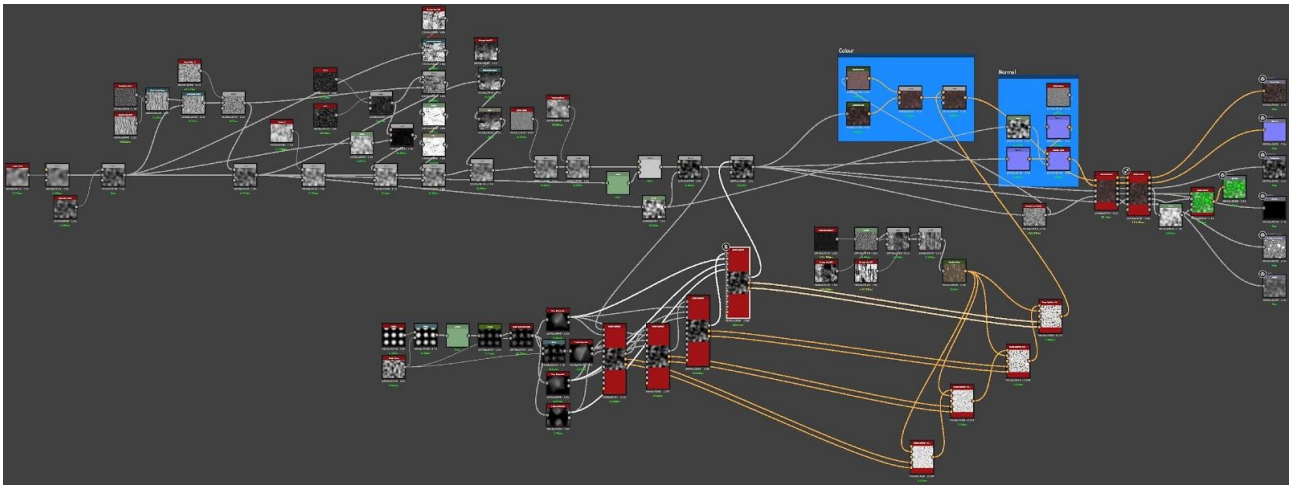



Pilt 2. Ülevaade Substance Designer töövaatest, milles on valmimas ühs mängu ehitise materjali tekstuurid.

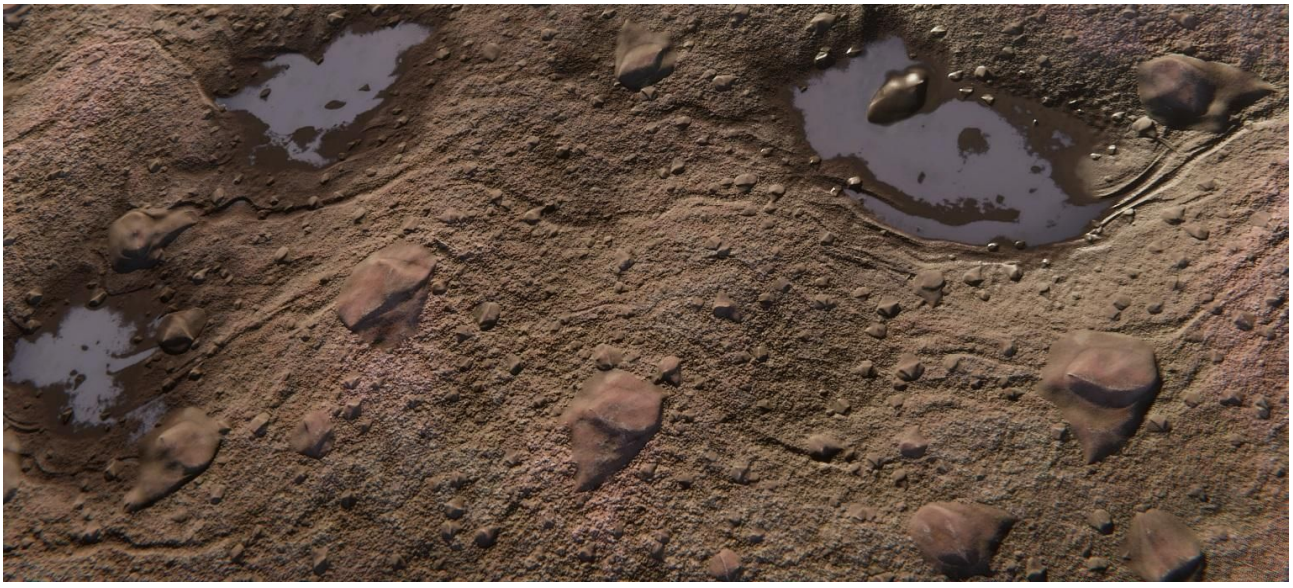
Peamisteks põhjusteks, miks eraldi graafe loodi (mitte ei loodud ühte suurt mitut materjali hõlmavat graafi), oli muutmälu kasutus, mis graafi töötlemiseks kulus. Näiteks kasutas 85 elemendist koosnenud graaf kuni 12 GB muutmälu. Teiseks põhjuseks oli graafide loetavus, mis võis elementide lisamisega kahaneda.

## 4. 4. 1. Pinnase materjali tekstuuride loomine, kasutades Substance Designer tarkvara

Antud näites loodakse viis erinevat tekstuuri, millest koostatakse materjal, mis kujutab liivast, märga pinnast (koos kividega). Antud materjal koosneb 85 elemendist, millest enamus on erinevad protseduurilise müra funktsioonid, näiteks Perlini ja Gaussian müra, mille väärtusi on muudetud ning omakorda kokku segatud.

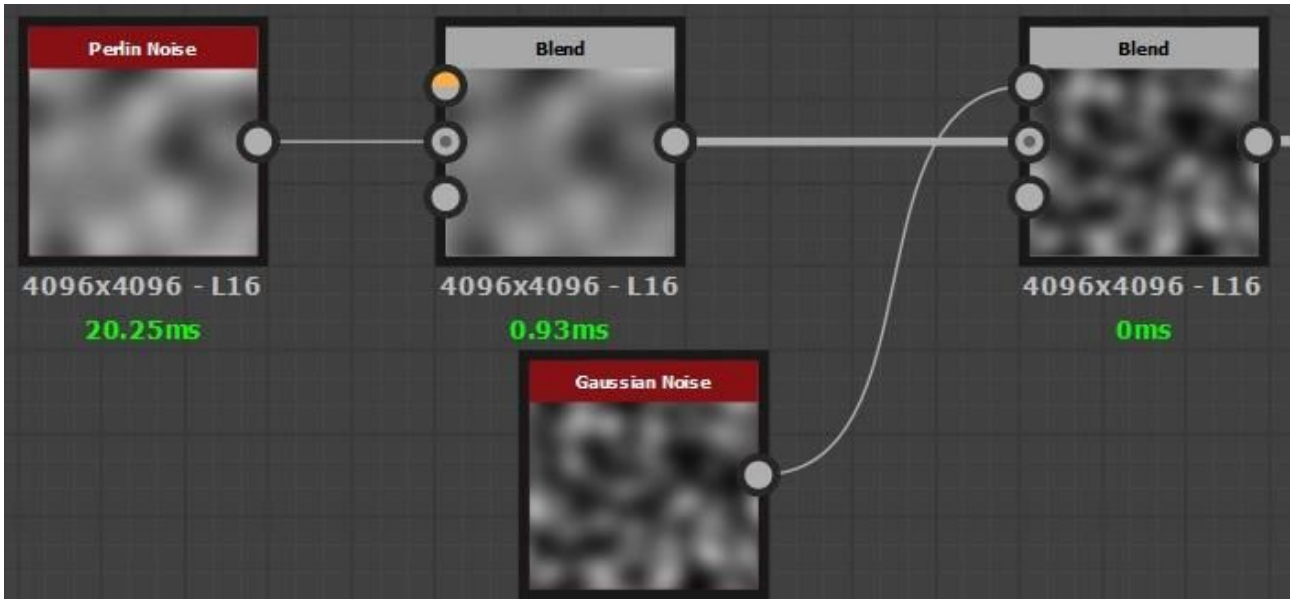


Joonis 8. Ülevaade Substance Designer'is loodavast materjalist



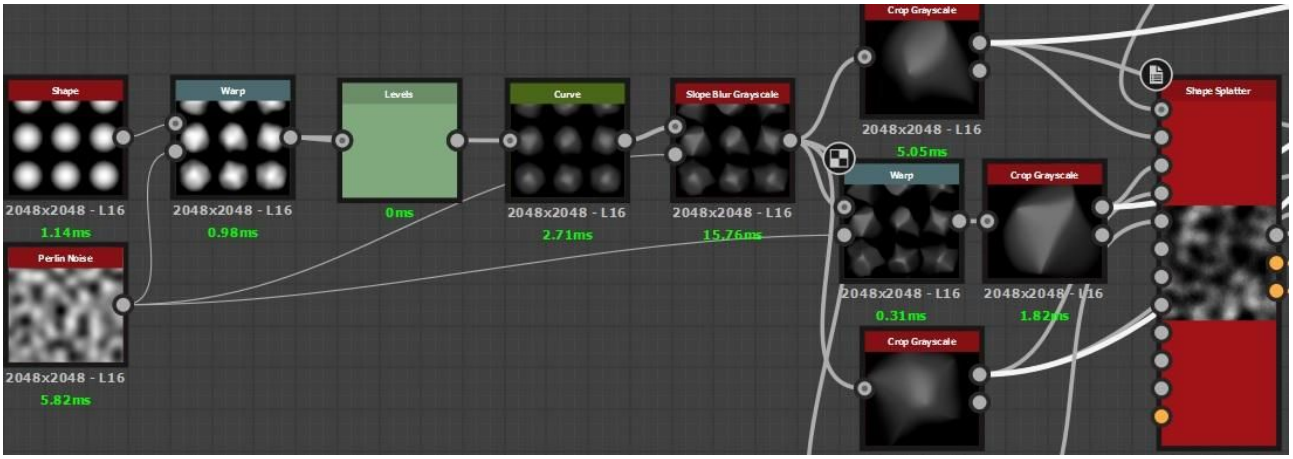
Pilt nr. 3. Unity's visualiseeritud materjal

Üleval oleval pildil on näha, milline on materjal kasutades graafi abil loodud tekstuuri. Teatava ruumilisuse loob pinna normaal kaart, mis muudab valguse peegeldumisnurka, vastavalt kaardil olevatele väärtustele.




Joonis 9. Graafil on näha, kuidas erinevad müra funktsioonid omavahel liidetakse. Antud pildil on elemendid olemuselt protseduurilise müra funktsioonid, kuid element võib omada ka keerukamat funktsiooni (näiteks muuta mustvalge elemendi värviliseks).

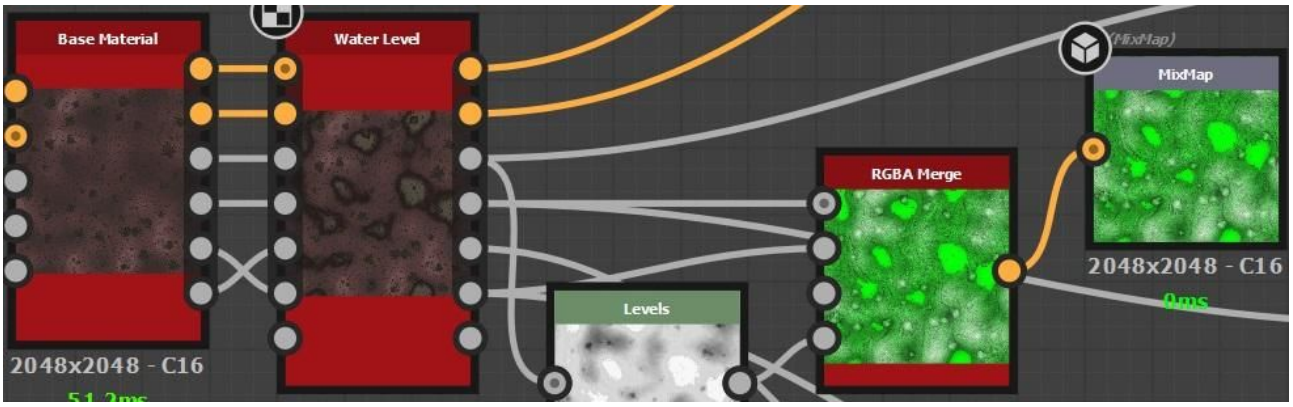
Lisaks on graafis kasutatud kujundite genereerimiseks ka lihtsaid põhikujusid, näiteks ringid, mille kuju on omakorda muudetud kasutades protseduurilist müra (joonis 10). Sellise lähenemisega on loodud materjalil olevad kivid. Neid kujundeid on kopeeritud viis tuhat korda, kasutades "Shape Splatter" elementi, mis muudab iga kopeeritava kujundi asukohta, suurust ja orientatsiooni (Substance Designer'i manuaal, 2019).




Joonis 10. Graafi osa, mille ülesandeks oli kivide lisamine materjalile.

Tekstuure on võimalik optimeerimise eesmärgil ka liita. Näiteks on võimalik kokku panna kõik mustvalget informatsiooni omavad tekstuurid. Arvutigraafikas leiab laialdast kasutust nelja kanali ehk RGBA kasutus ning sellises formaadis kasutab tekstuure ka Unity. Iga mustvalge tekstuuri on võimalik lisada ühe kanali alla. Näiteks on võimalik lisada metallilisuse tekstuur R kanali alla. Kõik neli kanalit on võimalik kokku pakkida üheks tekstuuriks, mida saab ära kasutada Unity Shader

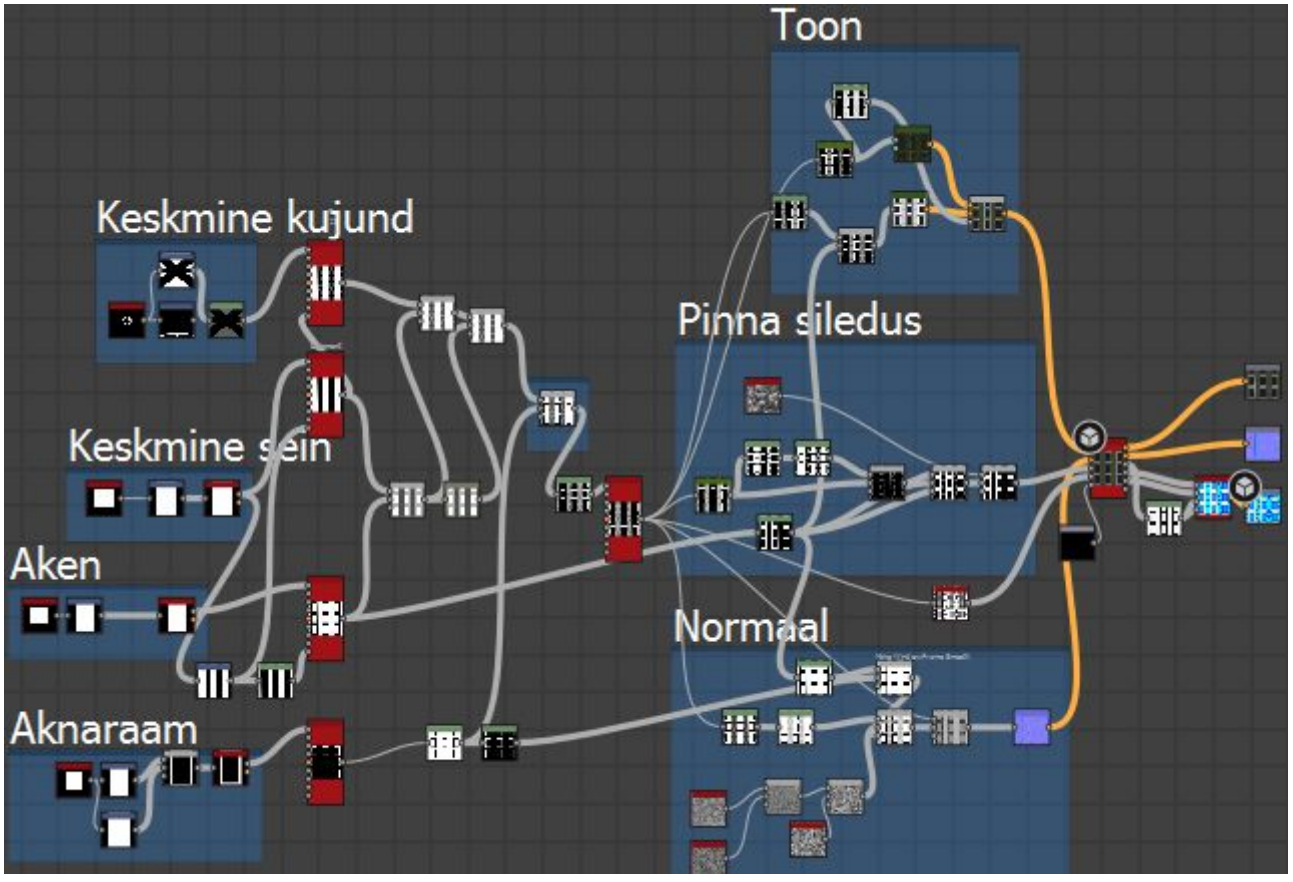
Graph lahenduses.




Joonis 11. Antud graafil on näha, kuidas liidetakse kanalid (RGA), et neid saaks optimeeritud materjalis kasutada.

## 4. 4. 2. Ehitise materjali tekstuuride loomine, kasutades Substance Designer tarkvara

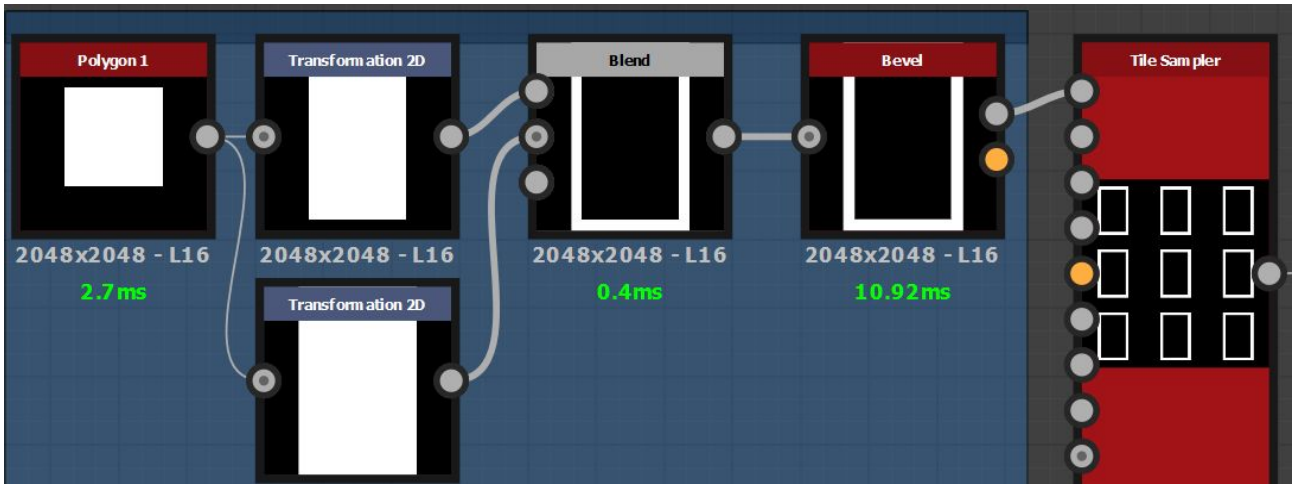
Substance Designer'it kasutades loodi ka ehitiste materjalid. Materjali loomise protsess oli sarnane pinnase materjaliga, mille tõttu selle loomisest täpsemat ülevaadet ei anta. Sarnaselt eelmise graafi koostamisele, kasutati ka siin protseduurilist müra. Näiteks leidis protseduuriline müra kasutust betooni meenutava tekstuuri osa loomise juures.




Joonis 12. Ehitise materjali loomine, kasutades Substanse Designer tarkvara.

Pilt 4. Ehitise materjal, mis on loodud kasutades Substance Designer tarkvara.

Erinevalt eelmisest materjalist, kasutatakse ehitise materjali loomisel väljalõikamise tehnikat ("Blend" elementiga). “Blend” sõlmpunkti abil on võimalik kasutada ühe elemendi pindala, et eemaldada teatud osa teisest elemendist. Seda tehnikat kasutati näiteks  aknaraami loomisel.




Joonis 13. Aknaraami loomine kasutades tarkvara Substance Desinger

Graafi parameetreid muutes oli võimalik genereerida erinevaid tekstuure. Näiteks muutes "Trasnformatsion 2D" elemendi väärtusi oli võimalik muuta akna kuju. Substance Designer lahendus võimaldas luua suurel hulgal materjale, mida linna keskkonna loomisel kasutada sai.

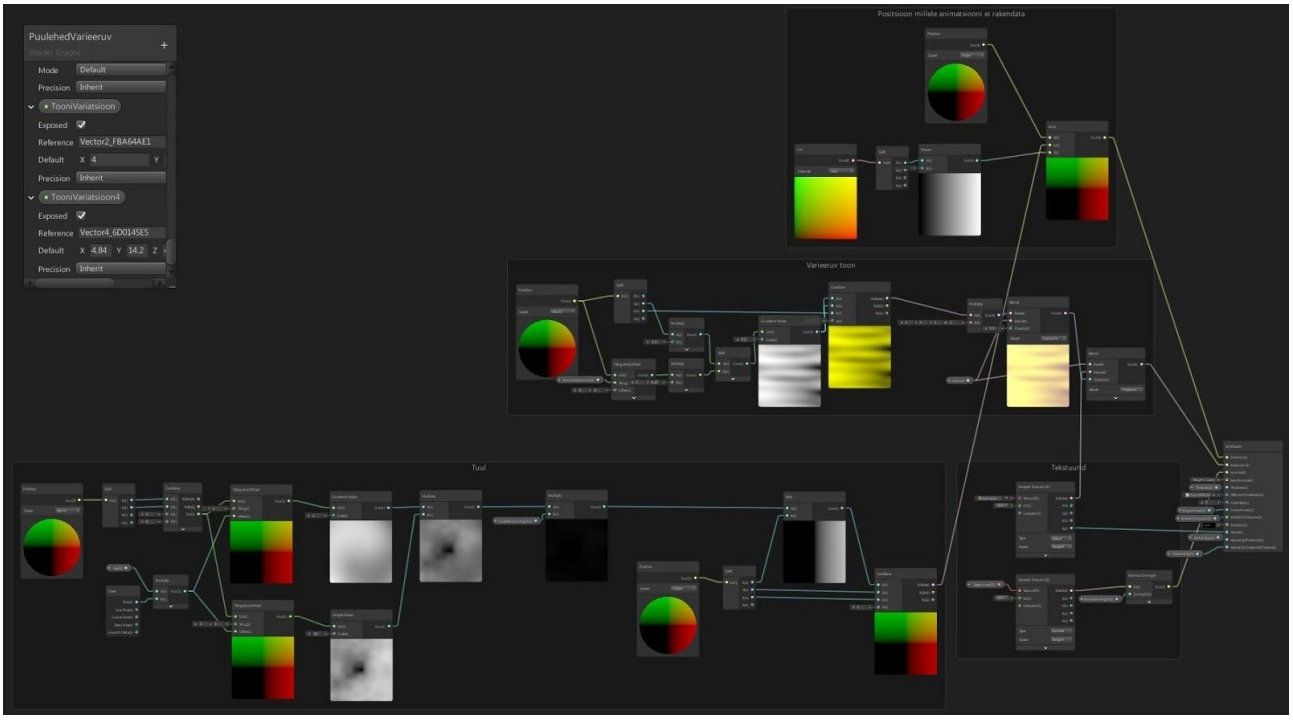
## 4. 5. Unity Shader Graph materjali loomise juures

Lõputöö käigus loodi unity shader graph abil 9 graafi, mida kasutati erinevate materjalide loomisel. 6 graafi tuginesed järgmises alapeatükis loodavale taimestiku materjalile, kuna erinevad taimed nõudsid teatavaid muudatusi materjalis, mis nõudsid omakorda graafikutes muudatusi (muutused, mida ei olnud võimalik saavutada muutujatele teise väärtuse andmisega). Shader Graph leidis kasutust ennekõike materjalide loomisel, mille abil objekte animeeriti (näiteks puulehed). Üks graaf oli seotud pinnasega, mille abil anti edasi pinnasel asuva vee liikumist. Ehitistega oli seotud kaks graafi. Esimese ehtitise graafi abil muudeti tekstuuri tooni väärtusi vastavalt aja möödumisele. Teise abil pandi teatud ehitise osad kumama vastavalt tekstuuri ja aja väärtustele.

### 4. 5. 1. Unity Shader Graph taimestiku materjali loomise juures

Antud peatükis loodakse materjal, mis simuleerib puulehtede liikumist ning loob nendes ka teatava tooni varieeruvuse (joonis 14). Graaf koosneb 35 elemendist ning 8 muutujast, millega saab näiteks tuule intensiivsust või lehtede värvust programmi töötamise käigus mõjutada. Antud materjali loomise põhjuseks oli taolise materjali puudumine Unity arenduskeskkonnas (ennekõike, kui kasutusel on HDRP) ning selle vajalikkus keskkonna loomise juures.

Graaf on jaotatud neljaks suuremaks osaks. Nendeks kõige suurem tegeleb puulehtede liigutamisega ehk püüab simuleerida tuult, mis puulehti mõjutab. Teine (graafi keskel) tegeleb teatava variatsiooni loomisega puude värvuses. Kolmas graafi osa (joonis 14 kõige ülemine osa) jätab puulehe (või puu oksa) paigale ehk määrab ära ala, millele animatsiooni (ehk tuule mõju) ei rakendata. Neljandas osas asuvad puulehe tekstuurid.

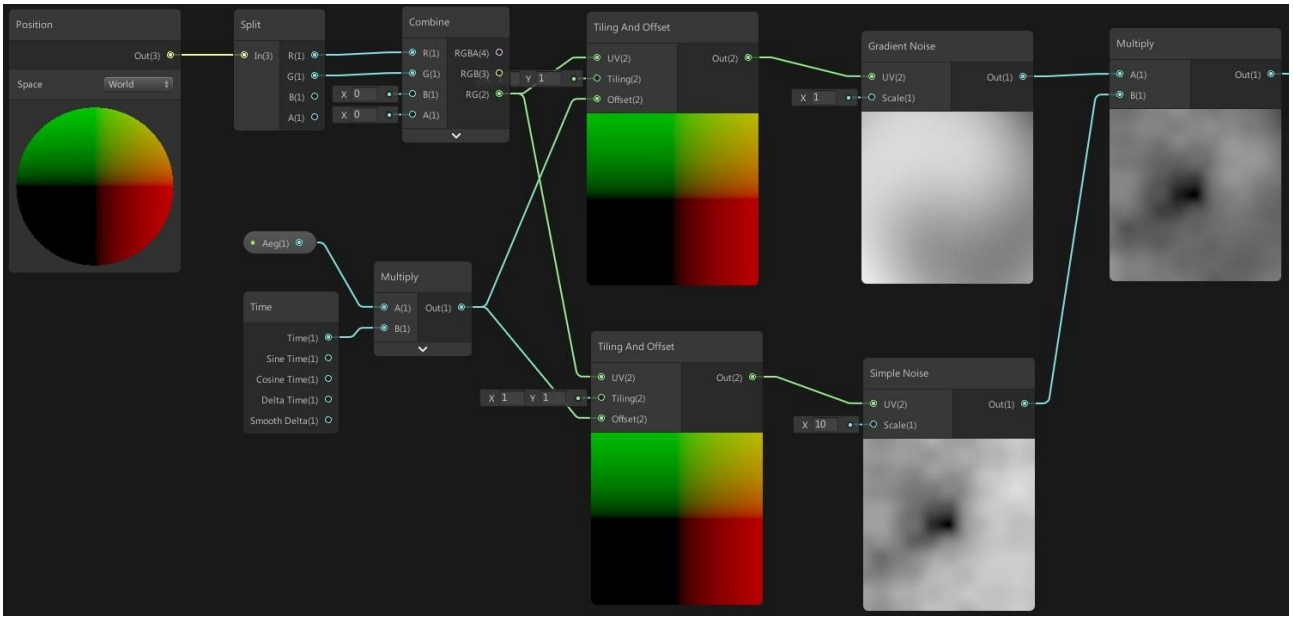


Joonis 14. Shader Graph näidis annab ülevaatliku pildi tervest koostatud graafist.

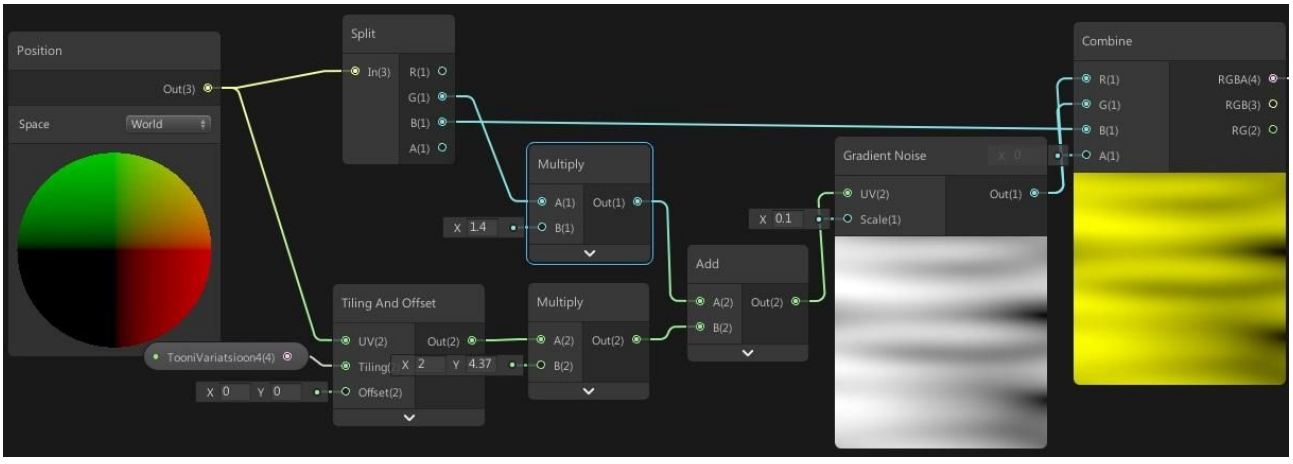
Lisaks on graafis muutujad, mis on liidetud erinevate graafi elementidega. Näiteks võib muutujate abiga mõjutada lehtede animatsiooni kiirust või tervet animatsiooni ning selle välimust. Lisaks on

ka mitmed materjali välimust mõjutavad tegurid seotud muutujate väärtusega, näiteks on võimalik mõjutada lehtede pinna siledust (ingl. *smoothness*) või lehtede värvust. Samuti on võimalik muuta kui puulehtede värvi varieeruvus astet või ala suurust, milles variatsioon toimub (kuna variatsiooni loob protseduuriline müra, siis muudetakse müra detailsust). Vastavalt muutuja väärtustele, võivad varieeruda üksikud lehed või terved puud.

Graaf on jaotatud kolmeks osaks, milles kõige suurem vastutab puulehtede animeerimise eest (joonis 15). Esmalt võetakse kasutusele positsiooni element, millest eraldatakse R ja G kanal. Seejärel liidetakse antud kanalid elementidega "Tiling And Offset", millele lisatakse aja element, mis vastutab animatsiooni kiiruse eest. Järgnevalt liidetakse mõlema “Tiling And Offset” elemendiga protseduurilise müra element. Kasutusel on antud elemente kaks tükki. Esimene on väiksema detailsusastmega ning teine tunduvalt detailsem. Visuaalse müra elementidega simuleeritakse puulehtede liikumist (müra element genereerib teatud arvulisi väärtusi, mis mõjutab verteksite asukohta). Vähem detailsem müra simuleerib suuremaid tuulepuhanguid ning detailsem annab edasi puulehtede kiiret liikumist.



Joonis 15. Tuule loomine kasutades protseduurilist müra.



Joonis 16. Puulehtede värvuse variatsiooni loomine.

Teine tähtis osa graafist tekitab puulehtedele teatava variatsiooni (joonis 16). Ühes graafi osas eraldatakse G, B kanalid (mida võib vaadelda kui telgede koordinaate). Teises osas liidetakse

positsiooni element Tiling And Offset elemendiga, mis määratleb kui detailne on värvi muudatus (näiteks võib variatsioon olla seotud lehtede või tervete puudega). Edasi võimendatakse esimese ja teise osa elementide väärtusi "Multiply" elemendiga. Seejärel liidetakse elementide väärtused kokku. Summa suunatakse protseduurile müra koordinaatide (UV) sisendiks. Seejärel liidetakse kanalid uuesti kokku elemendiga "Combine" (tulemus liidetakse hiljem puulehe tekstuuriga).

## 4. 6. Unity Unity VFX graph mängu loomise juures

VFX graafi kasutatakse mitmete mängu elementide loomise juures. Näiteks luuakse pilvesüsteem, mille välimust saab muutujate abil muuta. Veel luuakse VFX graafi abil süsteem, mis võimaldab lisada keskkonda suurel hulgal objekte, kasutades nende loomise kohaks punktikogumit. Objektideks võivad olla puud või kivid maastikul. Samuti võib selle abil lisada ka hoonetele täiendavat detailsust.

### 4. 6. 1. Pilvesüsteemi loomine

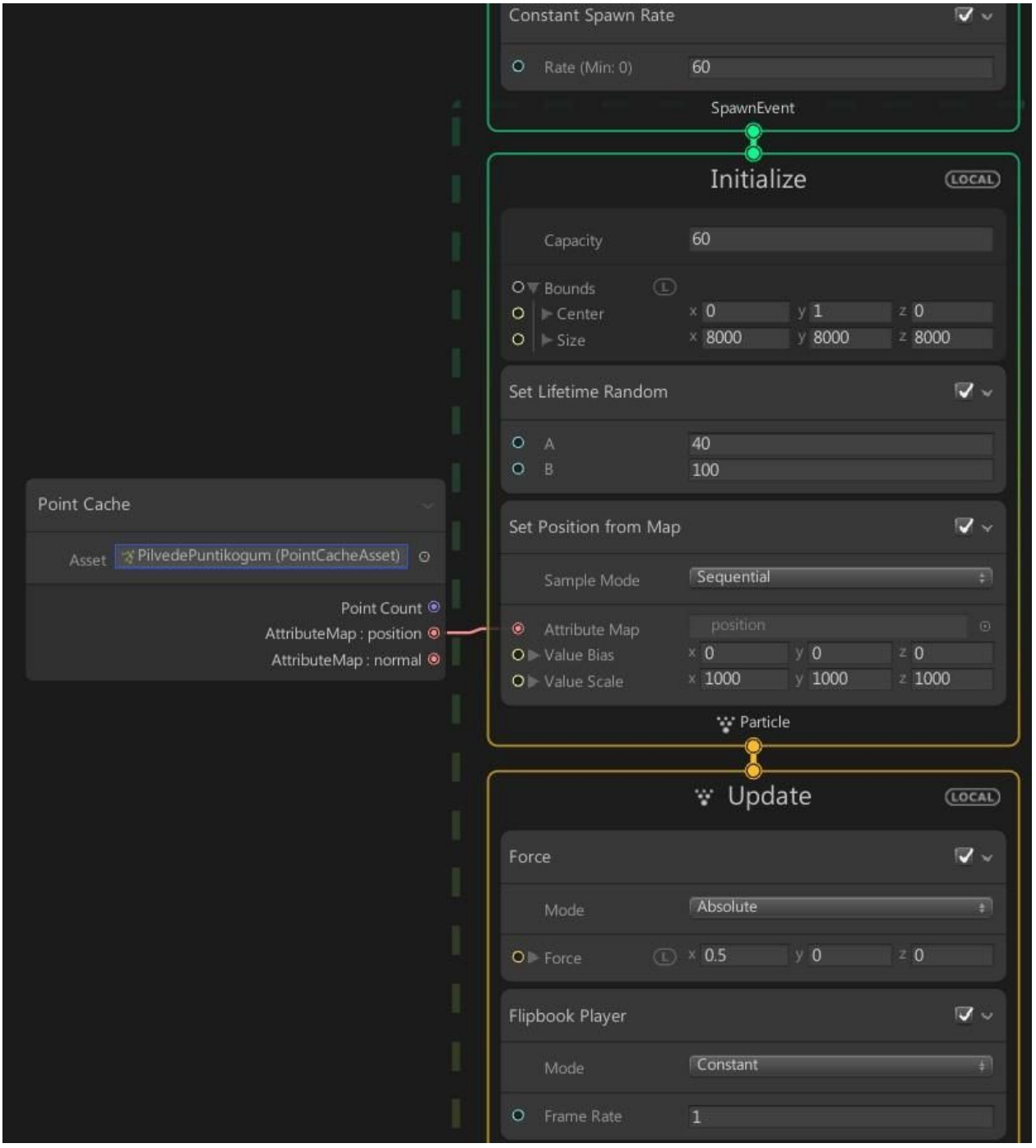
Antud graaf võimaldab aktiveerida ning liikuma panna suure hulga osakesi (ingl *particles*). Osakene võib olla ka pildijada, mis tuleb luua eraldi tarkvaralahenduses. Pildijadade (ingl *flipbook*) loomiseks on olemas mitmeid võimalusi ning ka nendele loomiseks sobivad tarkvaralahendused, mis on suunatud visuaalsete efektide loomiseks (näiteks Houdini FX). Antud töös võeti kasutusele universaale tekstuuride loomiseks mõeldud lahendus Substance Designer, mille abiga tekstuurid ning ka nendest moodustatud pildijadad luuakse.

Kuna terve pildijada peab samuti olema kantud ühele tekstuurile, kantakse üks pilt tektuurile 64 korda. VFX loeb pildijada igat pilti ning koostab sellest osakese, mis on vastavalt sellele ka animeeritud. Sellest tulenevalt on pildijada põhjal koostatud osakesed tunduvalt madalama resolutsiooniga, mis tihti eeldab kogu tekstuuri resolutsioon tõstmise vajadust.


Joonis 17. Pildijada koostamine kasutades tarkvara Substance Designer.

Joonisel 17 on näha pildijada, mida VFX graafis kasutatakse. Muutes moonutuse väärtust (näiteks 0.02 võrra) inkrementaalselt ning genereerides vastavalt sellele ka uusi tekstuure, on võimalik koostada pildijadasid. Genereeritud tekstuurid tuleb kanda 8 korda 8 ruudustikule, millest moodustub pildijada, mida VFX graaf saab lugeda.

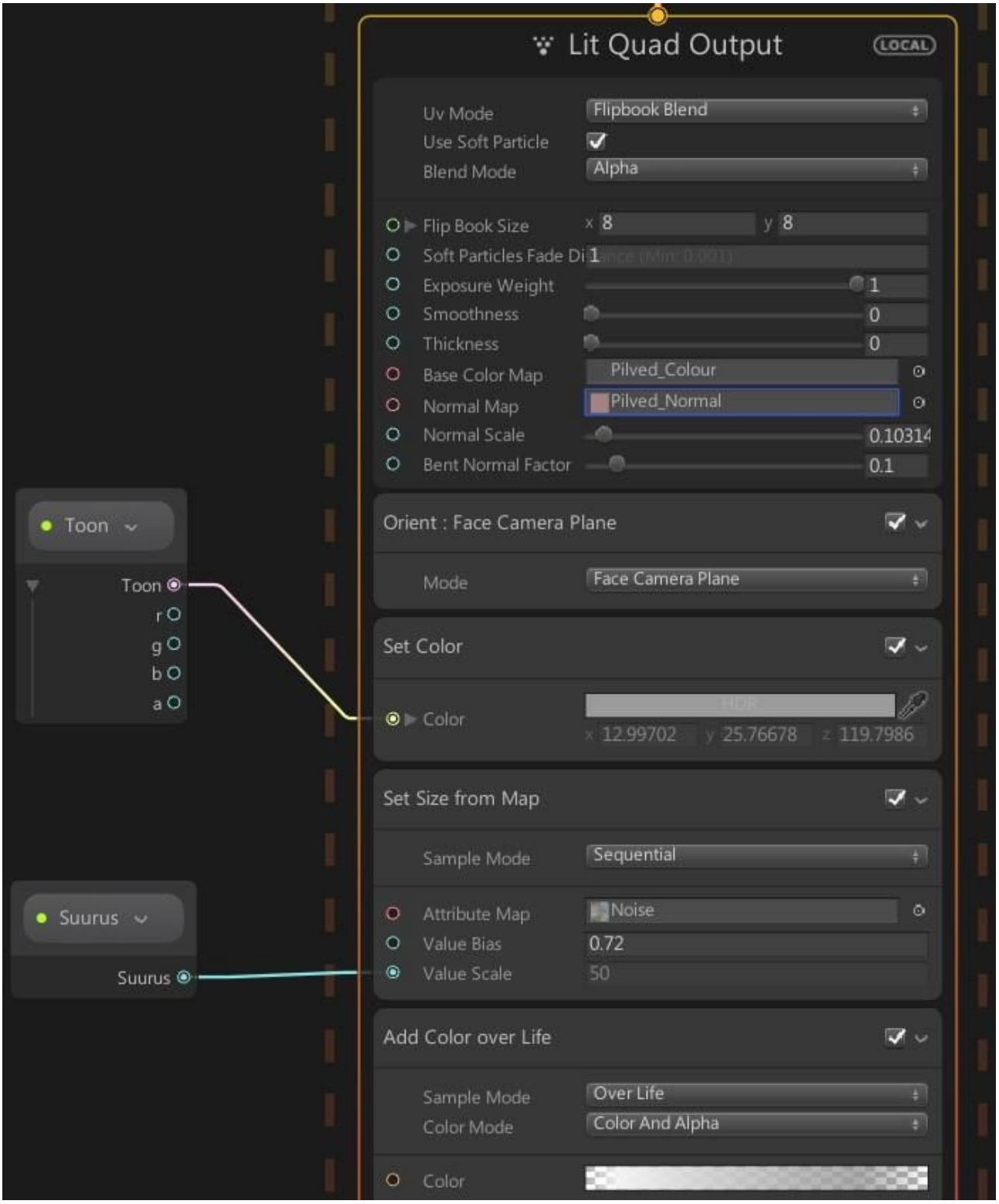
Pilvesüsteemi loomine algab "Spawn" kontekstis, milles plokk "Constant Spawn Rate" tekitab ajaühikus 60 osakest. Seejärel määratakse kontekstis "Initialize" maksimaalne osakeste arv, mis on süsteemis võimalik eksisteerida. Lisaks seatakse paika eksisteerimis ala suurus (“Bounds”), mis peab kaamera vaatevälja jääma, et osakesi kuvataks. Järgmisena antakse osakesele varieeruv eluaeg vahemikus 40-100 ajaühikut. Osakeste asukoht määratakse läbi punktikogumi. "Update" kontekstis määratletakse ära sellele rakendatava jõu suund ning ka ploki "Flipbokk Player"  kaadrisagedus.




Joonis 18. Pilvesüsteemi loomine.

Pildijada saab kasutusele võtta VFX graafi väljundi kontekstis (“Lit Quad Output”. Selleks saab kasutada Flipbook komponenti, mis pildijada loeb ning sellest animeeritud osakese loob (2D animatsioon). Väljundi kontekstis saab määrata, millist tekstuuri kasutatakse ning kui tegu on pildijadaga, saab määrata ka jada suuruse (antud näiteks on selleks 8 korda 8). Loodav pildijada on

suunatud alati kaamera poole ning selle värvust on võimalik muuta muutujaga "Toon". Suuruse variatsioon luuakse plokis "Set Size from Map", kasutades protseduurilise müra tekstuuri. Lisaks muutub pilvede värv ja läbipaistvus aja möödudes.



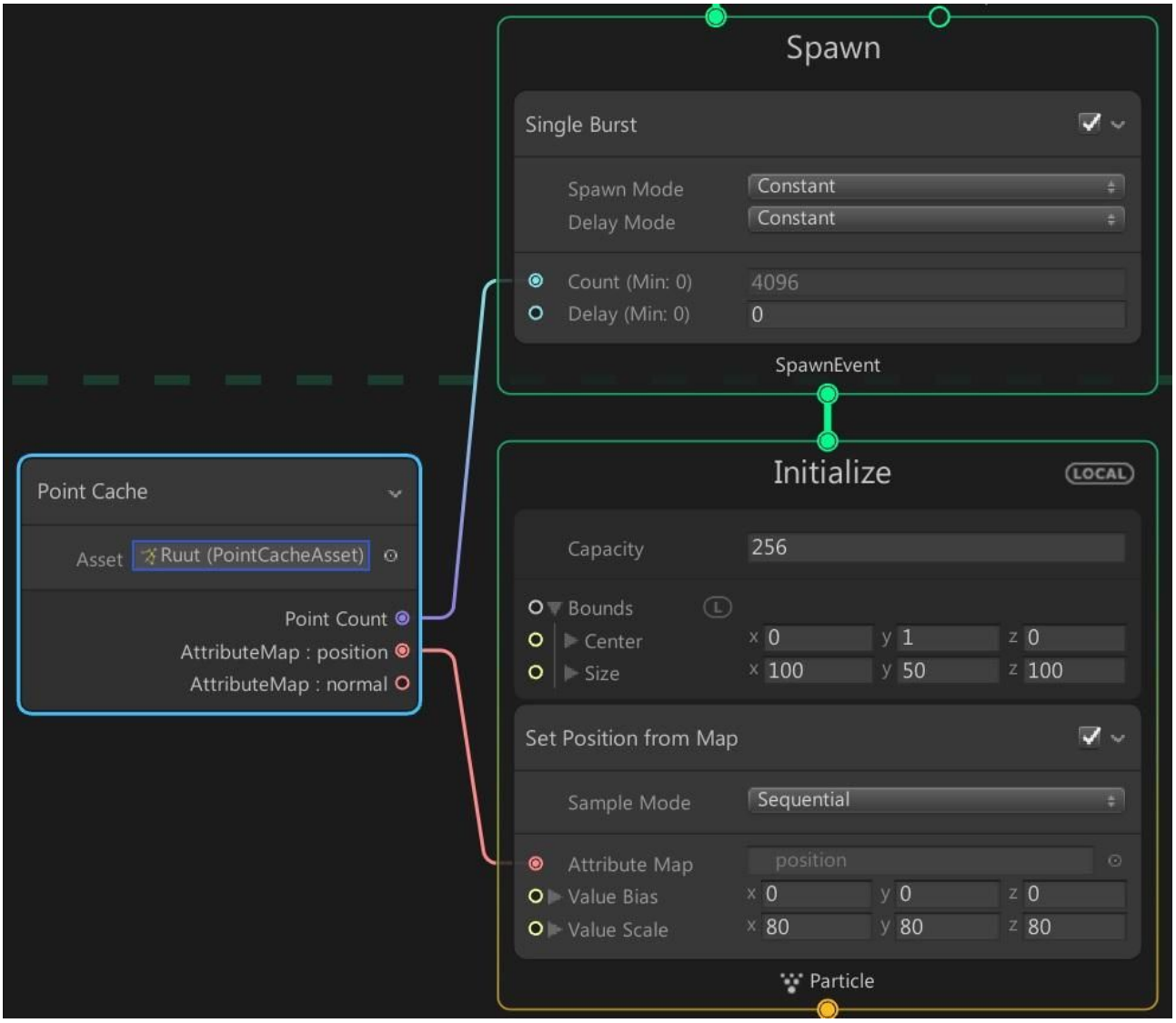
Joonis 19. Pilve süsteemi väljundi kontekst

Pilt 5. Pilvesüsteem

Pildil on näha valminud pilvesüsteemi (pildil on kaks pilvesüsteemi: ühes on heledad ning teises tumedamad pilved). Pilved on ka animeeritud vastavalt pildijadale. Pilved on suunatud ennekõike tausta elemendiks. Kuna optimeerimise eesmärgil on osakeste arv väikene ning iga osake on pööratud alati kaamera suunas, siis need ei sobi läbi lendamiseks (mis ei ole antud mängu kontekstis ka vajalik). Erinevaid parameetrite väärtusi kasutades saab ka pilvede välimust muuta.

## 4. 6. 2. Objektide lisamise süsteemi loomine kasutades VFX graafi

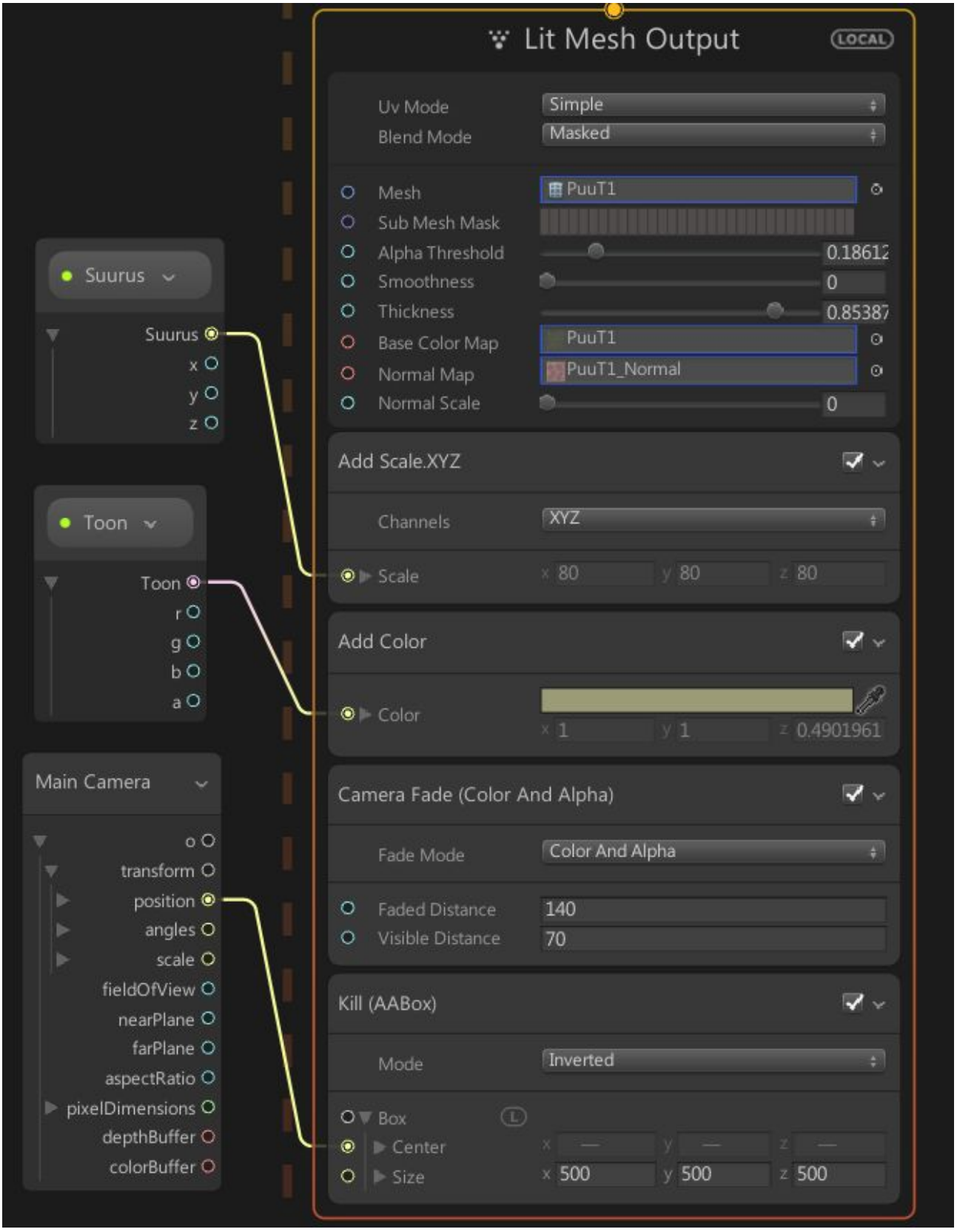
Loodav süsteem võimaldab lisada keskkonda suurel hulgal objekte, võttes koordinaadid ja objektide arvu, punktikogumi muutujast. "Spawn" kontekst tekitab objektid keskkonda. “Initialize” määrab, kui palju objekte võib antud süsteemis kokku eksisteerida (antud juhul 256, kuid see number võib olla tunduvalt suurem). Plokk "Set Position from Map" määratleb, kui suure ala sisse punkikogum mahutatakse.




Joonis 20. Objektide lisamise graafiku "Spawn" ja "Initialize" kontekst.

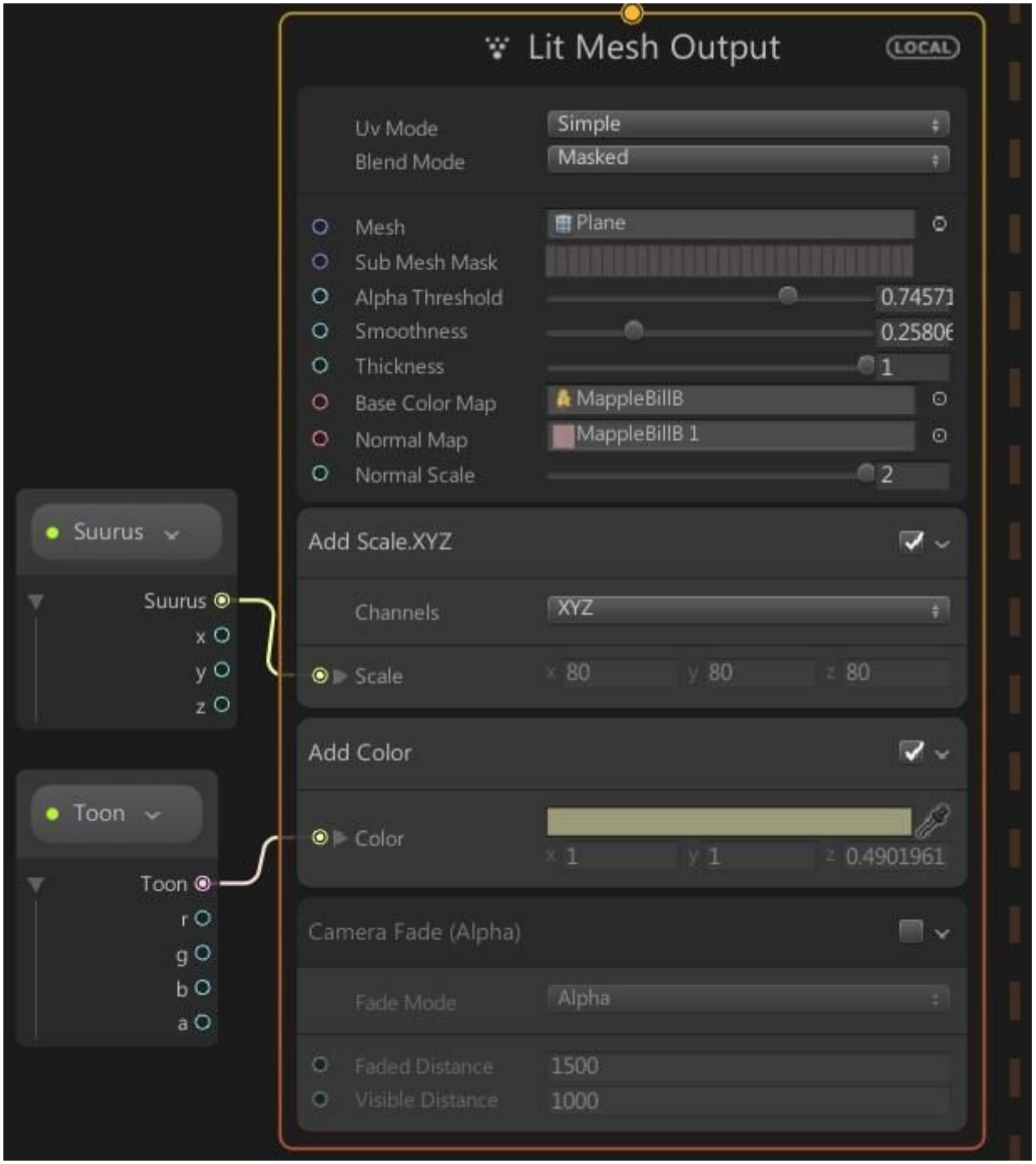
Väljundi kontekst (joonis 21) loob keskkonda detailse 3D objekti. Kontekstiga on lisatud kaks muutujat, mis määravad objekti suuruse ja värvuse, vastavalt kasutaja poolt sisestatud väärtustele. Konteksti on lisatud "Camera Fade" plokk, mis aitab objekti sujuvalt nähtavaks või nähtamatuks muuta (kuid ei lae objekte süsteemist välja), vastavalt kaamera kaugusele.

Kõige tähtsam osa graafist on kaamera muutuja, mis määratleb, kui kaugelt objekti kuvatakse. Plokk "Kill (AABox)" eesmärgiks on osakeste eemaldamine süsteemist (antud juhul eemaldatakse terve punktikogum). Selles graafis kasutatakse antud plokki osakeste nähtavaks tegemiseks (punktikogumi sisselaadimiseks) ehk näha on ainult objekte, mis asuvad kaamera objekti keskpunktist teatud kaugusel.



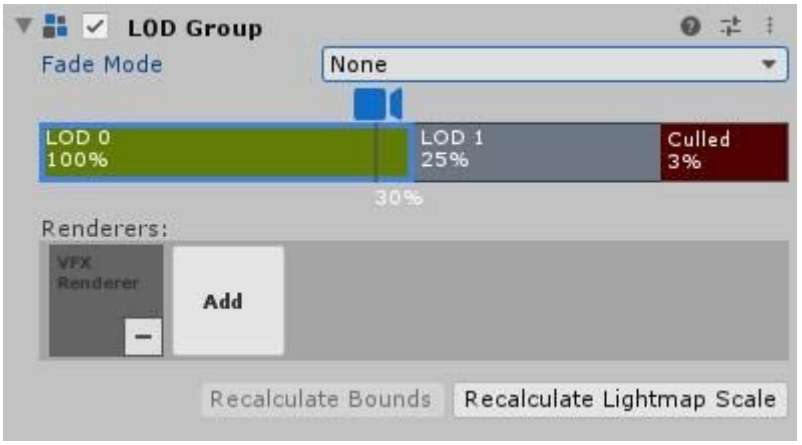
Joonis 21. Objektide lisamise graafiku väljundi kontekst.

Allolevat graafi osa võib vaadelda kui uue graafi väljundit. Kõik eelnevad graafi osad on jäänud samaks ning muudetud on ainult väljund. Antud väljund kuvab väga lihtsat 3D objekti, mis koosneb kahest risti asetsevast nelinurgast (ingl *imposter*). Kontekstile on lisatud kaks muutujat, mis määravad suuruse ja värvuse. Sellel väljundil võib rakendada ka "Camera Fade" komponenti, mis selle objekti (vastavalt kaamera kaugusele) ekraanilt ära kaotab (objekt muutub vastavalt kaamera kaugusele üha läbipaistvamaks).




Joonis 22. Väikese detailsusega objektide lisamise graafi väljund

Mõlemad graafid on võimalik siduda Unity LOD süsteemiga. Graafika optimeerimise eesmärgil asendatakse teatud kauguselt üks VFX graafi objekt teisega, vastavalt kaamera kaugusele (kaamera lähedal asub VFX graaf, mis loob detailsemaid objekte). Kuna graafis on kasutusel "Camera Fade" plokk, toimub üleminek ühelt (VFX) objektilt teisele sujuvalt ning LOD süsteemis ei ole vaja eraldi sujuvad üleminekut seadistada (selle tõttu on "Fade Mode" väärtus "None").



Joonis 23. VFX graafide LOD grupp.


Pilt 6. VFX graafi abil keskkonda lisatud puud.

Loodud süsteemi kasutati näiteks keskkonnas asetsevatele mägedele puude (Pilt 6) ning muude 3D objektide (kivide) lisamiseks. Mäest või selle osadest oli võimalik genereerida punktikogum, kasutades selleks Unity arenduskeskkonnas olevat töövahendit Point Cache Baker. Tänu sellele oli võimalik keskkonda lisada suurel hulgal detaile, küllaltki kiiresti. Kokku lisati keskkonda antud süsteemi kasutades 65000 objekti.

## 4. 7. Graafika optimeerimine

Selles peatükis antakse ülevaade graafika optimeerimismeetodite rakendamisest. Optimeerimise käigus rakendati Unity LOD süsteemi, mille eesmärgiks oli vähendada graafikaprotsessori poolt töödeldavate verteksite arvu. Lisaks rakendatakse HLOD süsteemi, mis kuulub DOTS lähenemise alla. HLOD võimaldab vähendada töödeldavate objektide ning materjalide (sh. tekstuuride) arvu, asendades mitmed väiksemad objektid teatud kauguselt ühe suurema objektiga. Kolmadaks püütakse optimeerida ekraanil toimuvat ülejoonistamist (Overdraw), mille vähendamise abil saab eeldatavalt langetada graafikaprotsessori tööd ja tõsta kaadrisagedust. Kõigi meetodite rakendamise mõju kaardisagedusele mõõdetakse erinevate mõõtvahenditega, milleks on Unity Profiler ja NVIDIA Nsight.

Graafika optimeerimise juures kasutatakse peale suuremate optimeerimismeetodite ka mitmeid, eeldatavalt vähem mõju avaldavaid, meetodeid. Näiteks on võimalik optimeerida Shader Graph'i abil loodavaid graafikuid. Kasutades protseduurilise müra asemel tekstuure on võimalik graafikaprotsessori koormust vähendada, kuna graafikaprotsessor ei pea genereerima protseduurilist müra, vaid peab lugema eelnevalt valmis tehtud tekstuuri (Sanglimsuwan, 2018). Kui asendada graafide avalikud muutujas staatiliste väärtustega on võimalik samuti vähendada graafikaprotsessori tööd (Sanglimsuwan, 2018). Lisaks liideti mitmed tekstuurid üheks, et graafikaprotsessoril oleks vajalik töödelda vähem tekstuure.

Optimeerimise meetodiks võib tuua ka selle, et seati sisse VFX graafiga loodud objektidele (sh. pilved), kaamera vaateväljaga seotud väljalaadimis süsteem. Näiteks määratleti (üljuhul ristkülikukujuline) ala, mis pidi jääma kaamera vaatevälja, et VFX graafi abil loodud osakesi kuvataks. Sellest tulenevalt ei kuvatud osakesi, mis ei jäänud kaamera vaatevälja.

### 4. 7. 1. LOD süsteemi rakendamine graafika optimeerimise juures

Esimene graafika optimeerimismeetod, mida töö koostamise käigus kasutati, oli detailsuse astmete loomine (LOD). Unity arenduskeskkonnas on kasutusel LOD süsteem, mis võimaldab moodustada LOD gruppe. Antud süsteem vahetab detailsemad objektid välja vähem detailsemate vastu vastavalt kaamera kaugusele objektist. Antud süsteem võimaldab ühelt detailuse astmelt üle minna sujuvalt, kuvades ülemineku ajal korraks mõlemad objekti (läbipaistvalt). LOD süsteemi kasutati pooltel ehitistel. LOD süsteemi peamiseks eesmärgiks on verteksite arvu piiramine, mida graafikaprotsessor peab töötlema. LOD süsteem ühildus ka Unity DOTS lähenemisega ehk see töötas ka eniteetidega (sujuv üleminek entiteetidega ei ühildunud). Lisades HLOD komponendi näiteks ehitisele, lisatakse koos sellega ka LOD grupp. Töös kasutati LOD süsteemi enamasti osana HLOD süsteemist. Erandiks olid VFX graafi objektid, milles rakendati ainult Unity LOD süsteemi.

Erineva detailsusega objektid loodi rakenduses Blender 2.80. LOD 0 on originaalne objekt, mis ehitise puhul küündis teatud juhtudel 10000 verteksi piirimaile. LOD 1 on näha kaugemalt ning seal jäi verteksite arv üljuhul vahemikku 100 - 1000. Ehitiste puhul oli LOD 2 enamasti ristkülik ning see laeti sisse, kui objekt hõlmas endas vähem kui 5 % ekraani pikslitest. Olenevalt ehitise suurusest, laetakse hoone lõpuks täielikult välja, kui seda on väga raske kaugelt märgata (ennekõike väiksemad ehitised).

### 4. 7. 2. HLOD süsteemi rakendamine programmi optimeerimise juures

HLOD süsteemi kasutati poolte ehitiste optimeerimiseks ehk iga ehitis millel oli enam kui 500 verteksit. Mõnedel ehitistel olid küljes täiendavad 3D objektid, mis omasid materjale. HLOD süsteemi kasutati antud objektide välja laadimiseks või nende kõigi asendamiseks ühe vähem detailsema objektiga (sh. nende liitmist hoone ja selle materjaliga). Peale ehitiste optimeeriti HLOD süsteemiga ka tervet keskkonda. Näiteks mõningad kaljud võisid koosneda mitmetest osadest ja omada erinevaid materjale. Teatud kauguselt laeti need välja ning asendati ühe suure objektiga, mis omas ühte materjali (antud objekti loomine eeldas lisatööd 3D modelleerijas Blender).

### 4. 7. 3. Ülekatte (overdrow) vähendamine

Ülejoonistamine ehk ülekate oli ennekõike seotud loodud keskkonnas asetsevate puude ja taimedega. Kuna puulehed olid verteksite arvu vähendamise eesmärgil enamasti liidetud ühe suure nelinurgaga (üheks oksaks) tuli rakendada lehe materjalil läbipaistvust (pilt 7). Antud ruut kujutab mängus ühte puu väiksemat oksa (mida võib olla puu küljes üle 500), mis on liidetud puu suurema 3D oksaga. Sellest tulenevalt võis olla enam kui 75% puulehtede materjalist läbipaistev.

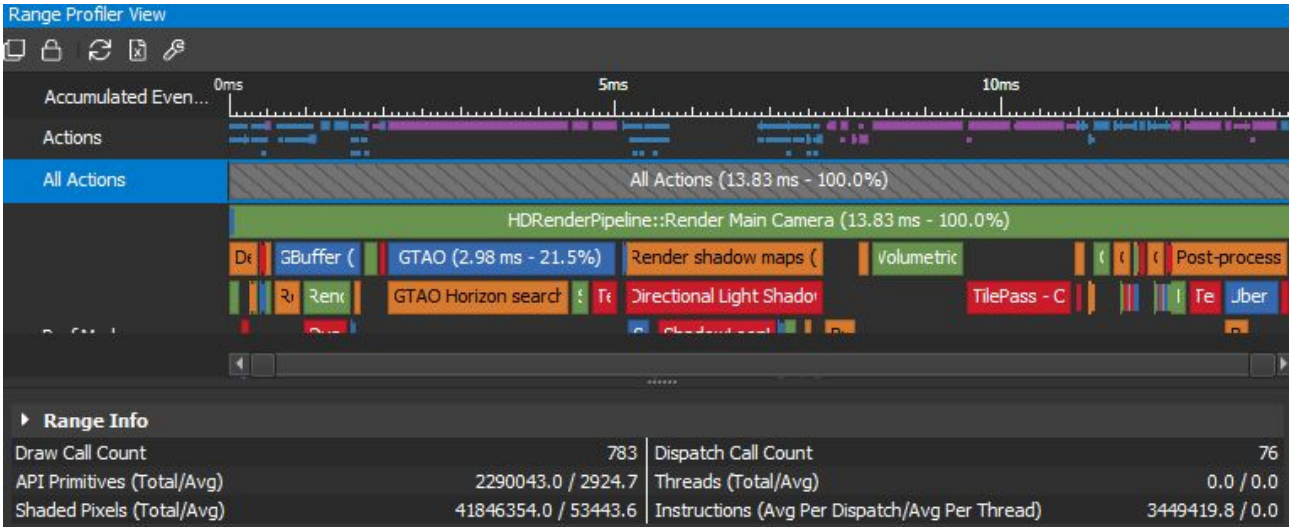
Pilt 7. Pildil on kujutab puuoksa materjali tekstuuri (materjal kantakse ruudu peale). Kogu kollane ala on loodavas mängus läbipaistev, et ruut jätaks lehtede mulje.


Pilt 8. Pilt pargist, mida optimeeritakse.

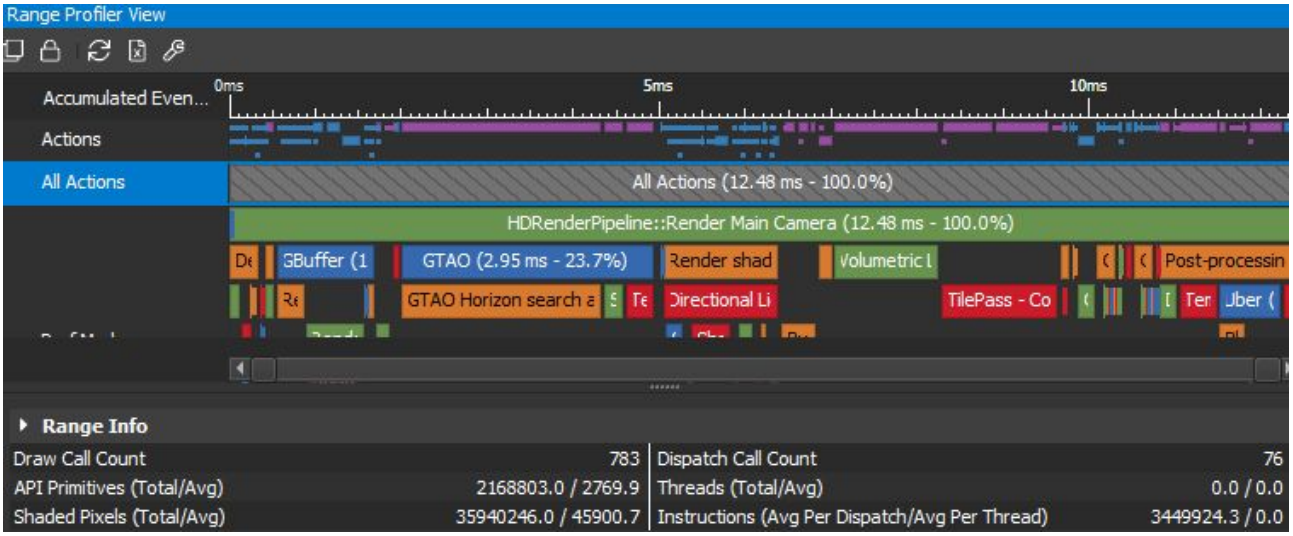
Üheks võimaluseks, kuidas vähendada läbipaistvat ala, on asendada ruut keerukama kujutisega. Näiteks on võimalik modelleerida kujund, mis omab väiksema oksa kuju. Antud meetodit optimeerimise käigus ka rakendati. Lisaks asendati 2D oks ehk ruut välja 3D oksaga, kus iga leht oli eraldi modelleeritud. Meetodite abiga tõusis puu verteksite arv mitmekordselt (teatud juhtudel kuni neli korda), kuid läbipaistva ala suurust, mis ülejoonistamise tekitab, vähendati. Kõigil puudel kasutati ülekatte vähendamiseks LOD süsteemi, mis püüdis ka suurenenud verteksite arvu mõju programmile vähendada. LOD süsteemi abiga vähendati puude lehti vastavalt kaamera kaugusele (kauguses kuvati ainult kolmest ruudust koosnevat 3D objekti).

Kuna mängu loomisel on kasutusel HDRP, ei ole võimalik ülekatte mahtu Unity arenduskeskkonnas testida, kuna selleks mõeldud töövaade on eemaldatud (Mcgrail, 2018). Sellest tulenevalt tuli kasutusele võtta arenduskeskkonna väline tööriist, milleks valiti Nvidia Nsight Graphics tarkvara, mis võimaldab ülekatet mõõta.

Ülekatte mahtu on võimalik mõõta muuhulgas kasutades Nvidia Nsight Graphics tarkvara, lugedes graafikaprotsessori poolt töödeldud pikslite (Shaded Pixels) arvu (Kiel, 2015). Antud tarkvara võeti ülekatte mõõtmisel kasutusele ning testiti loodud keskkonna erinevates vaadetes.




Joonis 24. Nsight Graphics (2019.3.1) Range Profiler vaade. Ülevaade optimeerimata pargi kuvamisega seotud kaadriajast. Graafikakaart: GTX 1060 6GB. Full HD (1080p) resolutsioon.




Joonis. 25. Ülevaade optimeeritud pargi kuvamisega seotud kaadriajast. Graafikakaart: GTX 1060 6GB. Full HD (1080p) resolutsioon.

Täiesti tühjas keskkonnas (ainult mäed) mõõdeti 20 miljonit töödeldud pikslit. Pilve süsteemi lisamisega kaasnes 3 miljonit töödeldud lisa pikslit. Viies mängus oleva kaamera vaate pargi sisse (kaamera vaateväljas oli 80 puud ning mäed), tuli töödelda kokku ligikaudu 41,8 miljonit pikslit (joonis 24), mis viitas suurele ülekatte tasemele ning mille tõttu otsustati seda ka optimeerida. Peale läbiviidud optimeerimise vähenes töödeldud pikslite arv 41,8 miljonilt 35,9 miljonile, mille abil vähenes ka kaadriaeg (joonis 25).

### 4. 7. 4. Kogu mängu keskkonna testimine

Antud peatükk annab ülevaate kogu keskkonna testimisest, kasutades erinevat riistvara. Mõlemas testis kasutati Unity versiooni 2019.2, paketti Entities 0.1.1, Hybrid Renderer paketti 0.1 ning kasutusel oli HDRP versioon 6.9.

Esmalt testiti keskkonda lauaarvutiga, millel on järgnev riistvara: GeForce GTX 1060 6GB, Core i7 950, 14 GB RAM. Testimisel kasutav resolutsioon: 1080p (Full HD). Kasutati Unity Profiler ning Profiler Analyzer tööriistu.

Testid toimusid läbi kahe vaadete. Esimeses vaates oli näha terve linna koos pilvede ja VFX graafiku abil keskkonda lisatud puudega. Lisaks oli ekraanil ka mägine maastik. Esimeses kaamera vaatevälja jäid kõik 15000 lisatud ehitist, millest 7500 oli lisatud HLOD süsteem ning kõigile “Box Collider” komponent. Lisaks oli vaateväljas pilvesüsteem ning 65000 VFX graafiku abil lisatud puud (mis koosnesid kuuest kolmurgast).

| Name | Depth | Median | Media |
|---|---|---|---|
| PlayerLoop | 1 | 16.29 | |
| PostLateUpdate.FinishFrameRendering | 2 | 7.70 | |
| UpdateFunction.Invoke() | 3 | 7.46 | |
| GC.Collect | 5 | 7.35 | |
| PresentationSystemGroup | 2 | 7.29 | |
| Default World Unity.Entities.PresentationSystemGroup | 4 | 7.29 | |
| Default World Unity.Rendering.RenderMeshSystemV2 | 5 | 7.17 | |
| WaitForJobGroupID | 4-10 | 7.12 | |
| RenderPipelineManager.DoRenderLoop_Internal() | 3 | 6.83 | |

Joonis 26. Unity Profiler Analyzer, milles mõõdetakse "Render Thread" tööaega. Kaamera asetseb linna ääres ning selle vaatevälja jäävad kõik ehitised.

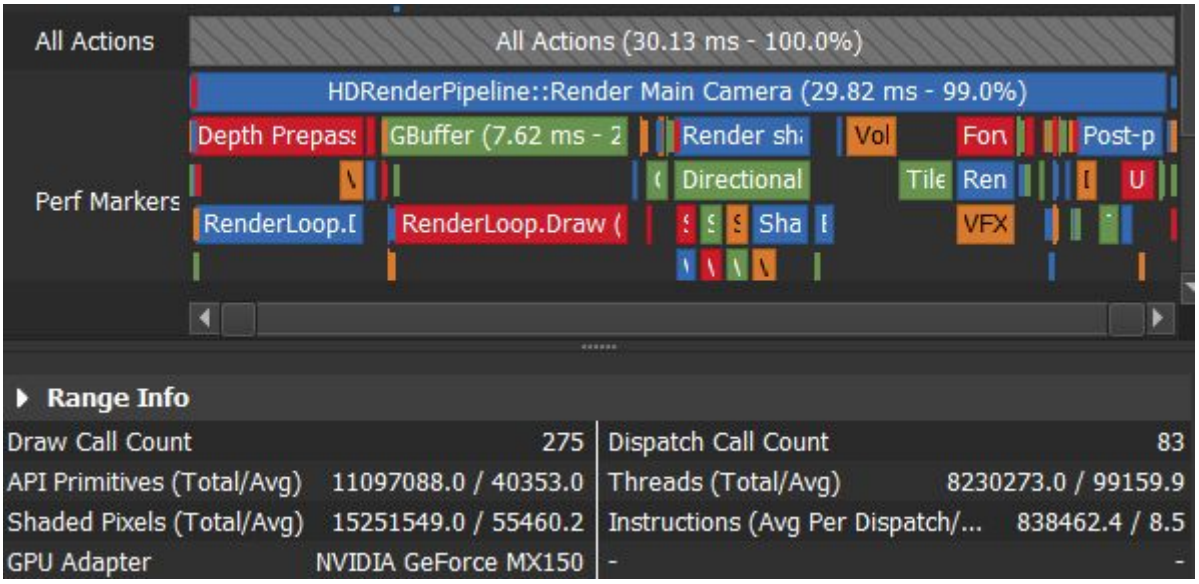
Teine vaade oli linnas olevast pargist, milles kaamera ees oli 80 puud. Park asus linna keskel. Vaatevälja jäi neljandik ehitistest. Ekraanil oli ka pilvesüsteem ning VFX süsteemiga lisatud puud.

| Name | Depth | Media | Media |
|---|---|---|---|
| PlayerLoop | 1 | 15.24 | |
| UpdateFunction.Invoke() | 3 | 7.11 | |
| PostLateUpdate.FinishFrameRendering | 2 | 6.98 | |
| PresentationSystemGroup | 2 | 6.95 | |
| Default World Unity.Entities.PresentationSystemGrou | 4 | 6.94 | |
| Default World Unity.Rendering.RenderMeshSystemV: | 5 | 6.81 | |
| RenderPipelineManager.DoRenderLoop_Internal() | 3 | 6.81 | |

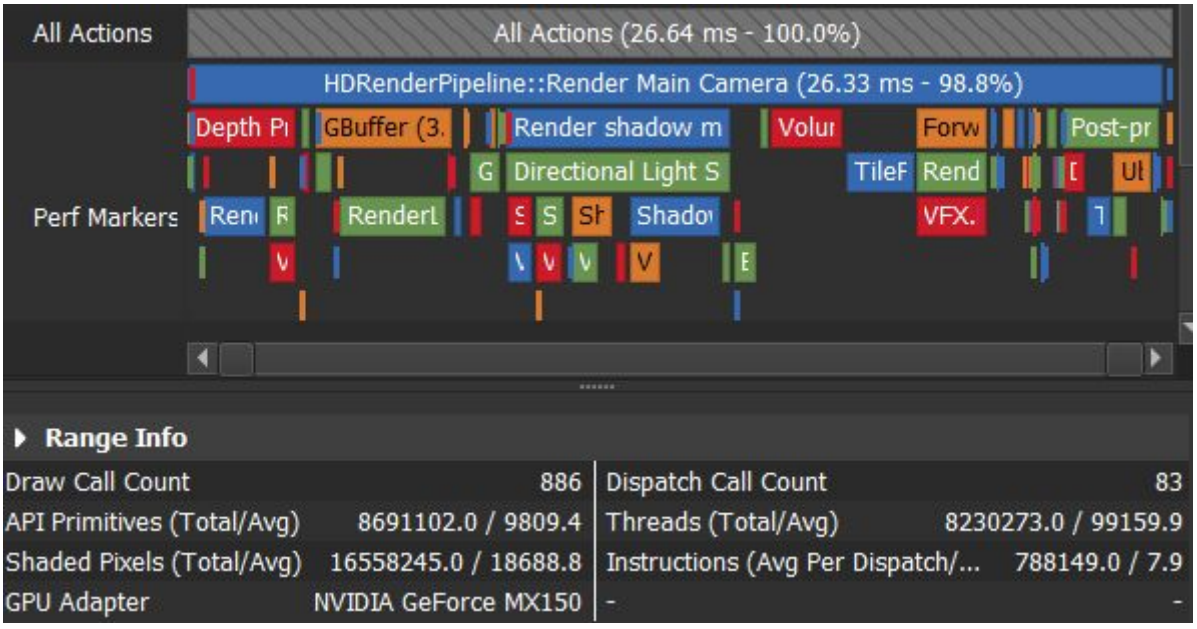
Joonis 27. Unity Profiler Analyzer, milles mõõdetakse "Main Thread" tööaega. Kaamera asetseb linna keskel asuvas pargis.

Tehtud testidest osutus kõige kulukamaks lõimeks "Main Thread", mille kaadriajast joonised 26 ja 27 ülevaate annavad. Kõige kulukam protsess "PlayerLoop" võttis vaates, mis hõlmas kogu linna, aega 16.29 millisekundit. Pargi vaates oli kaadriajaks 15.24ms, millest võib järeldada, et kõige rohkem võtab kaadriaega kogu linna hõlmav kaamera vaade.

Lisaks testiti keskkonda ka teise süsteemiga (sülearvuti) : Nvidia GeForce mx150 2GB, Core i3-8130U, 4GB RAM. Antud süsteemi testiti tarkvaraga Nsight Graphics 2019.4. Vaated olid samad, mis eelmise süsteemiga (testitav keskkond oli sama) ning resolutsiooniks 720p (HD).




Joonis 28. Nsight Graphics vaade, milles mõõdetakse kogu keskkonna kaadrisagedust.




Joonis 29. Nsight Graphics vaade, milles mõõdetakse pargis asetseva kaamera kaadrisagedust.

Kaadriaeg terve linna vaates oli 30.13 ms ning pargi vaates 26.64 ms. Testide tulemustest võib järeldada, et ka antud riistvaraga on kõige kulukam siiski tervet linna hõlmav vaade, isegi kui pargi vaates on töödeldavate pikslite arv suurem.

Testide tulemuste põhjal võib eeldada, et kaadriaeg vastas esitatud nõuetele ning sellest tulenevalt on loodud lahendused sobilikud linna ehitamise mängus kasutamiseks. Erinevatest kaamera vaadetest testimine oli vajalik, kuna kuna kaadreag võib varieeruda vastavalt mängija asukohale. Suurim kaardiajaga oli vaade, milles oli näha kõik ehitised.

## 5. Loodud graafiliste lahenduste tulevik

Loodud graafilised lahendused leiavad reaalset rakendust linna ehitamise mängus, mis suunatakse müügiplatvormile Steam. Näiteks aitab VFX graafi abil loodud 3D objektide lisamise süsteem lihtsustada ja kiirendada edasist mängu arendamist. Kuna loodud graafilised lahendused olid kooskõlas töö alguses seatud nõuetega, võetakse ka antud lahendused kasutusele loodavas mängus .

Tänu täidetud eesmärkidele on selge, et algne visioon küllaltki suuremahuliselt simulatsiooni mängust on võimalik ellu viia. Lisaks ei tule suurendada mängu miinimum-nõudeid riistvarale, mis annab võimaluse rohkematel inimestel loodavat tarkvara kasutada.

Töös leidsid kasutust ka mitmed eksperimentaalsed Unity laiendused, millest enamik (VFX Graph, HDRP) saavad mängus rakendamise küpseks eeldatavalt selle aasta lõpuks. Mõningad laiendused, näiteks ECS, liidetakse järgmisel aastal üha uute Unity mehhanismidega (näiteks animatsioonid). Selle tõttu tuleb ka oodata mängu välja andmisega, kuna eksperimentaalsed laiendused võivad langetada mängu stabiilsust. Siiski olid kõik laiendused piisvalt stabiilsed, et neid rakendada ning testida, mis tõttu sai need loodava mänguga ka integreerida. Kõik eksperimentaalsed lahendused olid vajalikud soovitud graafika ning kaadrisageduse saavutamisel. Lisaks andis ECS visuaalse programmeerimise lahenduse töös rakendamine võimaluse võtta antud lahendus kasutusele ka ülejäänud loogika koostamisel, mis võib oluliselt kiirendada mängu arendamist ka tulevikus.

Unity toob versioonis 2021.1 välja ka uue globaalse valgustuse (ingl *global illumination*) lahenduse (Mortensen. 2019), mis parandab eeldatavalt mängu üldist valgustust (ennekõike valguse peegeldumine ühelt pinnalt teisele). Kuna graafikaprotsessori kaadriaega jäi üle, saab tulevikus kaalutleda ka uute graafiliste lahenduste lisamisest mängule.

# 6. Kokkuvõte

Lõputöös anti teoreetiline ülevaate Unity arenduskeskkonnas olevatest lahendustest sh. andmetele orienteeritud tehnoloogiate kogumist (DOTS) ning Shader Graph ja VFX graafist. Samuti anti teoreetiline ülevaade visuaalsest programmeerimisest ning kuidas see on mängude arendamise juures rakendust leidnud.

Töö praktilises pooles loodi linna ehitamise mängu keskkond ning linna ehitamise mehaanika prototüüp, mida optimeeriti kasutades ECS lähenemist. Antud lähenemise abiga optimeeriti näiteks ehitiste kuvamist.

DOTS-i abil sooritatud optimeerimise mõju programmi kiirusele mõõdeti Unity Profiler ja Profiler Analyzer vahenditega. Lõputöös anti ka ülevaade testide tulemustest. Kokkuvõtvalt võib öelda, et kaadriaega, mis kulus objektide (ehitiste) kuvamiseks vähendati DOTS-i abil ligikaudu kolm korda.

Shader Graph abiga loodi mitmeid materjale, mida kasutati näiteks ehitistel ja maastikul. Ülevaade anti materjali loomise protsessist, mis algas tekstuuride loomisega Substance Designer'is ning nende abil materjali üles ehitamisega Shader Graph rakenduses.

VFX graafi abil loodi lõputöös pilvesüsteem, mis võimaldas dünaamilist muuta taeva välimust. Töös anti ülevaade selle süsteemi koostamisest ning kuidas on süsteemis olevate muutujate abiga võimalik väljundi ehk pilvede välimust oluliselt muuta.

Lõputöö eesmärgiks oli luua mängu keskkond ning sellel olevad graafilised lahendused, mis pidid töötama sujuvalt ka miinimum nõuetele vastava riistvaraga. Keskkonna loomine eeldas ka graafilise poole loomist, võttes arvesse ajalisi kui ka riistvaralisi piiranguid, mis oli samuti üheks osaks eesmärgi täitmise juures. Programm ei tohtinud olla limiteeritud protsessori ega graafikaprotsessori poolt. Kuna mängu visuaalne pool sai valmis ning vastas töökiirusele seatud kriteeriumitele (kaadrisagedus), võib lugeda eesmärgid täidetuks.

## Kasutatud allikmaterjalid:


ans_unity, 2019. DOTS Visual Scripting first experimental drop.
https://forum.unity.com/threads/dots-visual-scripting-first-experimental-drop.677476/

Ante J., 2018. ECS Track: Welcome and General Overview. Unite LA
https://www.youtube.com/watch?v=7ons0eVjhDY

Cooper T., 2018.Introduction to Shader Graph: Build your shaders with a visual editor.
https://blogs.unity3d.com/2018/02/27/introduction-to-shader-graph-build-your-shaders-with-a-visual-editor/

John O' Reilly, 2018. Creating explosive visuals with the Visual Effect Graph.
https://blogs.unity3d.com/2018/11/27/creating-explosive-visuals-with-the-visual-effect-graph/

Khalil I., 2019. C# Job System + ECS usage and demo with Intel. Unite LA.
https://www.youtube.com/watch?v=fp1D45hhVEM

Kiel J., Kerschner R., 2015. Brighter & Faster Graphics with NVIDIA Nsight VSE 4.5. GDC 2015.
http://developer.download.nvidia.com/assets/events/GDC15/GEFORCE/NsightVSE_GDC15.pdf

Kieran C., 2018. The High Definition Render Pipeline: Getting Started Guide for Artists.
https://blogs.unity3d.com/2018/09/24/the-high-definition-render-pipeline-getting-started-guide-for-artists/

Mcgrail A., 2018. Missing scene shading modes.
https://forum.unity.com/threads/missing-scene-shading-modes.561421/

Mortensen J., 2019. Enlighten will be replaced with a robust solution for Baked and Real-time GIobal Illumination.
https://forum.unity.com/threads/enlighten-deprecation-and-replacement-solution.697778/

Sanglimsuwan S. , 2018. Best practices for Shader Graph. Unite LA.
https://www.youtube.com/watch?v=Y6WfgFI5H90

Substance Designer, 2019. Substance Designer User Guide.
https://docs.substance3d.com/sddoc/substance-designer-user-guide-102400008.html

thierry_untiy, 2019. DOTS Visual Scripting 2nd experimental drop.
https://forum.unity.com/threads/dots-visual-scripting-2nd-experimental-drop.696500/

Unity Technologies , 2019. Enities manual. Entity Component System
https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/index.html

Unity Technologies, 2019. Burst manual. Burst User Guide.
https://docs.unity3d.com/Packages/com.unity.burst@1.1/manual/index.html

Unity Technologies, 2019. Jobs manual. Unity Jobs Package
https://docs.unity3d.com/Packages/com.unity.jobs@0.1/manual/index.html

Unity Technologies , 2019. Unity User Manual (2019.2).
https://docs.unity3d.com/Manual/index.html

Unity Technologies, 2019. Visual Effect Graph manual.
https://docs.unity3d.com/Packages/com.unity.visualeffectgraph@6.9/manual/index.html

Unity Technologies, 2019. Shader Graph manual.
https://docs.unity3d.com/Packages/com.unity.shadergraph@6.9/manual/index.html

Unity Technologies, 2019. Unity Scripting Reference. Scripting API.
https://docs.unity3d.com/ScriptReference/

**Lisa 1. Litsents**

**Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks**

Mina, Ahti Maa,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose
   Graafika loomine ning optimeerimine arenduskeskkonnas Unity,

mille juhendaja on Margus Luik,

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.

*Ahti Maa*

***14.08.2019***