TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

2-mõõtmeliste valgusefektide teegi projekteerimine ja realisatsioon

Bakalaureusetöö

Üliõpilane: Jaanus Varus
Üliõpilaskood: 104232IABB
Juhendaja: lektor Raul Liivrand
Kaasjuhendaja: Kaarel Allik

Tallinn
2015

Autorideklaratsioon

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

---------- ----------

(*kuupäev*) (*allkiri*)

Annotatsioon

Bakalaureusetöö eesmärk on leida lahendus probleemile, kuidas simuleerida lihtsamaid valgusefekte 2-mõõtmelisel tasandil, kusjuures põhirõhk on täis- ja poolvarjude loomisel. Töö tulemiks on *MonoGame* raamistikul loodud üldkasutatav teek.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 58 leheküljel, 6 peatükki, 36 joonist, 16 tabelit.

Abstract

The goal of the Bachelor's thesis is to find a solution to a problem of how to simulate simple 2D lighting effects in real time. Main focus is on the generation of umbra and penumbra regions caused by blocking a light source. The result of this thesis is a general purpose library built on top of MonoGame framework.

The thesis is in Estonian and contains 58 pages of text, 6 chapters, 36 figures, 16 tables.

Lühendite ja mõistete sõnastik

C# ***C Sharp***
Programmeerimiskeel, mis austab tugevaid tüüpe, imperatiivset, deklaratiivset, funktsionaalset ja objektorienteeritud programmeerimisdistsipliine.

HLSL ***High-level Shading Language***
Programmeerimiseel, mis võimaldab kirjutada C-süntaksiga programme, mis käivitatakse läbi Direct3D graafikakonveieri graafikaprotsessoril.

Stseen ***Scene***
Graaf (andmestruktuur), mida kasutatakse graafilistes rakendustest elementide loogiliseks jaotamiseks.

Gaussian blur ***Gaussian blur***
Pilditöötlusmeetod, mis ähmastab pilti kasutades Gaussi funktsiooni.

Scissor test ***Scissor test***
Operatsioon, mis praagib välja pikslid, mis jäävad *scissor rectangle*'ist väljapoole. Kuna seda operatsiooni tehakse ainult telg-orienteeritud ristkülikukujulise regiooni peal, siis selle testi sisendstruktuuri nimetatakse *scissor rectangle*'iks.

Tekstuur ***Texture***
Arvutigraafikas sõrestikmudelile rakendatud pilt, mis visualiseerimisel annab mudelile sellise välimuse, nagu oleks tal tekstureeritud pind.

Valguskaart ***Lightmap***
Tekstuur, mida kasutatakse valgusefektide loomiseks stseenile.

Raamistik ***Framework***
Loodavat platvormi, tarkvara, riistvara, protokolli vms toetav struktuur.

Graafikakonveier ***Graphics pipeline***
Leidub graafikakaardis ning koosneb mitmest aritmeetikaplokist või mitmest protsessorist, mis teostavad üldlevinud visualiseerimisoperatsioonide (perspektiivprojektsioon, akende kärpimine, jne) erinevaid osi.

Sujutamine ***Blending***
Arvutigraafikas kahe või enama objekti omavaheline pikslikaupa kombineerimine, näit. kahe pildi sujuv ühendamine või ühe pildi peale teise, läbikumava pildi paigutamine.

Rasterdamine ***Rasterizing***
Rasterdamist teostab rastriprotsessor (RIP), mis muudab teksti ja pildid pikslitest koosnevaks maatriksiks.

Sügavus-šabloontest ***Depth-stencil test***
Graafikakonveieri etapp, milles praagitakse välja pikslid, mis kukuvad läbi sügavustesti või šabloontesti.

Indie mäng ***Indie game***
Arvutimäng, mis on loodud ilma publitseerija rahalise toetuseta.

Jooniste nimekiri

Tabelite nimekiri

Sisukord

1. Sissejuhatus

Valgus on üks olulisemaid aspekte graafilistes rakendustes - see loob atmosfääri. Korrektselt valitud valgustusmeetodite abil saab manipuleerida vaataja poolt stseenist omandatavat infot ning tema emotsioone.

1.1 Taust ja probleem

Graafika programmeerimises (näiteks graafiliste arvutimängude arendamisel) tuleb kokku puutuda otsusega, kas ja mil moel simuleerida valgust ja selle suhtlust käsitletava stseeni objektidega. Kuigi valguse simuleerimine on küllaltki uuritud valdkond, võib teatud esteetilise kvaliteedi saavutamiseks materjale nappida.

Autori inspiratsiooniallikas antud teema valida oli arvutimäng *Dark*, mille eripäraks on just esteetiliselt kaunis valguse ja varjude mäng, kuid mille valgussüsteemi lähtekood ei ole avalik ega mängu autori poolt detailselt kirjeldatud. [1] Analoogsed avatud lähtekoodiga valgustuse lahendused simuleerisid poolvarje kasutades näiteks täisvarjude peal pilditöötlusmeetodit *Gaussian blur* või ei hallanud olukordi, kus valgusallikas oli varje heitvast kehast oluliselt suurem nii, et keha külgedest heidetud poolvarjud lõikusid (vt peatükk 2.2.3). [2] [3] Autori poolt leitud allikad ei pakkunud sarnast kvaliteeti nagu seda tegi mäng *Dark*.

Antud töö leiab lahenduse küsimusele, mil moel simuleerida reaalajas valgusallikaid ning valguse ja objektide koosmõjul tekkivaid varje, kusjuures eriline rõhk on lisaks täisvarjudele ka poolvarjudel ning nende loomisel tekkivatel probleemidel. Luuakse teek *MonoGame* raamistikul kasutades keeli *C#* ja *HLSL*.

Töö tulemusest saaksid kasu 2-mõõtmeliste graafiliste programmide arendajad, kellele antud funktsionaalsus vajalik võiks olla. Olgu selleks kasuks teadmine, mil moel töös käsitletud süsteemi luua või võimalus ise kasutada töö valmimisel loodud teeki, mida luues on arvesse võetud eelkõige töökindlust, kuid ka jõudlust ja avaliku programmeerimisliidese kasutusmugavust.

Graafiliste rakenduste arendamisega puututakse kokku neile suunatud tarkvarafirmades, aga ka ülikooli projektides või iga indiviidi isiklikust huvist (nagu see on autori näitel).

1.2 Ülesande püstitus

Antud töö on kirjutatud eesmärgiga leida lahendus probleemile, kuidas simuleerida 2-mõõtmelisel tasandil valgusefekte ja eelkõige valguse ja stseeni objektide koosmõjul tekkivaid täis- ja poolvarje.

Töös oodatavaks tulemuseks on *MonoGame* raamistikul loodud teek, mille abil kasutajal on kerge lisada oma rakendusse eelmainitud valgusefekte. Lisaks peaks teek võimaldama infot, kuidas näeb välja taolise valgussüsteemi töövoog ning algoritmid valguse ja varjude simuleerimiseks.

1.3 Metoodika

Eesmärkideni jõutakse läbi teegi projekteerimise ja realisatsiooni. Seal juures võetakse arvesse järgmiseid aspekte:

1. **Töökindlus**. Süsteem ja selle komponendid peavad toimima tõrgeteta ning arvesse võtma äärejuhte ja ebasobivaid kasutajasätteid.

2. **Kasutajamugavus**. Teegi integreerimine olemasolevasse süsteemi peaks olema lihtne ja kasutajasõbralik ehk hõlmama tööle saamiseks võimalikult vähe tegevussamme. Lisaks peaks kasutajaliides olema iseennast kirjeldav.

3. **Kiirus**. Teegi teostus peab tõsiselt arvesse võtma jõudlust, kuna tegemist on süsteemiga, mis töötab reaalajas ja mille loogikat jõustatakse üldjuhul 30 või 60 korda sekundis.

1.4 Ülevaade tööst

Töö esimeses osas pannakse paika teegi poolt võimaldatud funktsionaalsus. Tuuakse välja soovitud efektid koos illustratsioonidega ning kirjeldatakse efektilt oodatavaid kasutaja poolt manipuleeritavaid omadusi.

Töö teises osas käsitletakse konkreetset teegi realisatsiooni *MonoGame* raamistikul. Tuuakse välja teostuse detailid ning dokumenteeritakse programmeerimisliidest. Põhjendatakse erinevaid disainivalikuid.

Töö kolmas osa tegeleb arendusvaatega. Tuuakse välja, mis jäi antud töö loomisel realiseerimata ning mis oleksid võimalikud edasiarendused tulevikus.

Töö viimases osas tuuakse välja teegi võimalikud rakendused ning integreeritakse see ühe *MonoGame* raamistiku ametliku näidismänguga. Selguvad vajalikud tegevused taoliseks integratsiooniks.

2. Funktsionaalsus (projekteerimine)

Järgnevalt on esitatud nõuded funktsionaalsusele, mida töös loodav teek peaks võimaldama. Suures plaanis tegeletakse valgusallikate ja kestadega. Kest iseloomustab keha (hulknurka), millest valgus läbi ei tungi. Selle nähtuse tagajärjel tekivad varjud. Rääkides varjudest, eristatakse täis- ja poolvarje.

2.1 Valgus

Valgus on elektromagnetkiirgus, samamoodi nagu seda on ka näiteks gammakiirgus või raadiolained. Antud töö kontekstis mõeldakse valguse all nähtavat valgust ehk lainepikkusvahemikku, mis on inimsilmale tajutav (kiirgus, mis jääb infrapuna- ja ultraviolettkiirguse lainepikkuste vahepealne). [4]

Valguse omadusi saab kirjeldada erinevates kontekstides. Kui füüsik on huvitatud valguse füüsikalistest omadustest (kiirus, polarisatsioon), siis kunstnik huvitub pigem valguse esteetilistest omadustest (valgustus, värvid).

Käesolev töö huvitub esteetilistest omadustest ning jääb suhteliselt kaugele füüsikas käsitlevast valgusteooriast. Tehakse mitmeid lihtsustusi. Valguse levikut kujutletakse kiirtena ning huvitutakse põhiliselt valguse värvi omadustest. Vaatluse alt jäävad välja nähtused nagu valguse peegeldumine ja murdumine. Samuti ei tegeleta valgustatavate objektide materjaliste omadustega. Küll aga tuuakse arendusvaates välja võimalikud arengud pinnanormaalide kaasamises stseeni, mille põhjal on võimalik jätta kasutajale 2-mõõtmelisest stseenist 3-mõõtmelise stseeni mulje (vt peatükk 4.1).

2.1.1 Välisvalgus (ambient light)

Välisvalguse all mõeldakse üldist või keskkonnast tulenevat valgust (näiteks päike, kuu jms), mis ei ole tingitud konkreetsetest stseeni elementidest. Seda kujutatakse kui valgusena, mille kiired on lõpmatult põrganud objektide pindadelt, valgustades kogu stseeni ühtlaselt.

Välisvalguse jaoks defineeritakse tema kasutaja poolt muudetavaks omaduseks ainult värvus. Valge värvus ehk täielik valgustatus ei tohiks muuta stseeni lõppvärvust/väljanägemist võrreldes stseeniga, kus valgussüsteemi ei kasutata. Musta värvuse korral oleks tegemist täieliku valguse puudumisega ehk ükski objekt ei oleks nähtav/eristatav.

2.1.2 Kohtvalgus

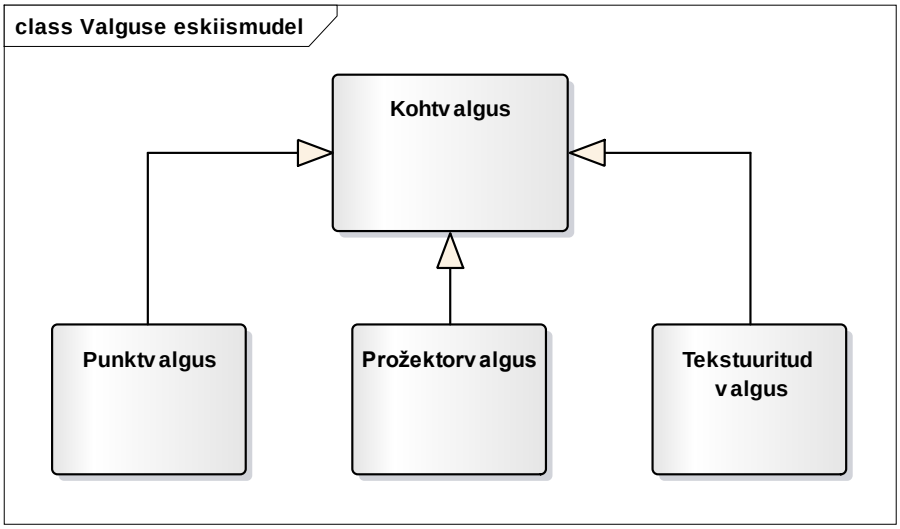
Kohtvalguse all peetakse silmas valgusallikat, mis on osa stseeni graafist (näiteks latern, taskulamp, jne). Taolised valgusallikad aitavad stseeni „ellu tuua“ ja mängivad peamist rolli soovitud atmosfääri loomisel.

Kohtvalgusallikatele defineeritakse järgnevad omadused koos kirjelduste ja piirangutega. Koordinaatidest rääkides peetakse silmas Cartesiuse koordinaatsüsteemi.

Tabel 1. Kohtvalgusti omadused

Omadus	**Kirjeldus**	**Piirangud**
Asukoht	Määrab ära valgusallika asukoha simuleeritava maailma koordinaatruumis.	
Rotatsioon	Määrab ära, kuhu valgusallikas on suunatud. Oluline eelkõige prožektor-tüüpi valgustitel.	
Suurus	Määrab ära valgusallika suuruse ehk kui kaugele ulatuvad valguskiired.	Suurusmõõdud ei tohi olla negatiivsed.
Värvus	Määrab ära valgusallika poolt erituva valguse värvi.	
Intensiivsus	Määrab ära valguse poolt erituva värvi intensiivsuse.	Intensiivsuse määr ei tohi olla negatiivne.
Raadius	Määrab ära valgusallika raadiuse ehk kui suure valgusallikaga (osaga, mis eritab valgust) on tegemist. Vajalik täis- ja poolvarjude arvestamisel.	Raadius ei tohi olla negatiivne.

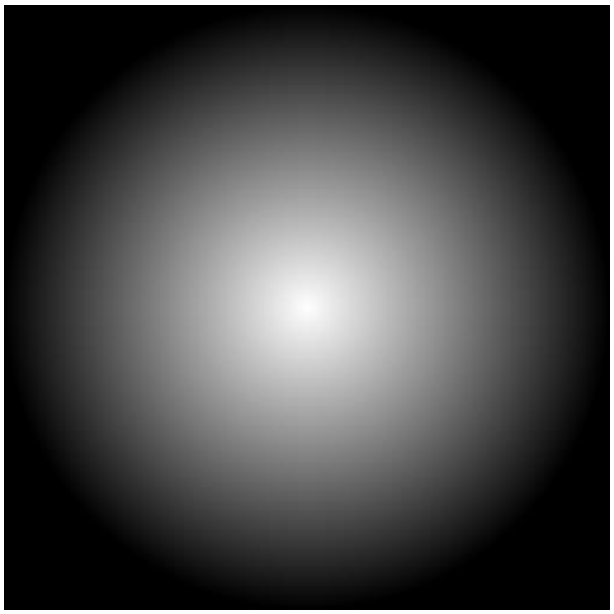
Antud töö raames eristatakse kolme tüüpi kohtvalgusallikaid: punktvalgus, prožektorvalgus, tekstuuritud valgus.



Joonis 1. Valguse eskiismudel

Punktvalgus kujutab endast isotroopset valgusallikat. See kiirgab ühtlaselt valgust igas ümbritsevas suunas (näiteks latern). Oluline on, et teek suudaks punktvalgust genereerida ilma, et kasutaja peaks omapoolselt võimaldama ressursse selleks (näiteks tekstuuri). Piirdutakse lineaarse hajuvusega valgusallika keskmest.

Punktvalgusele ei defineerita ühtegi omadust lisaks kohtvalguse üldistele omadustele.



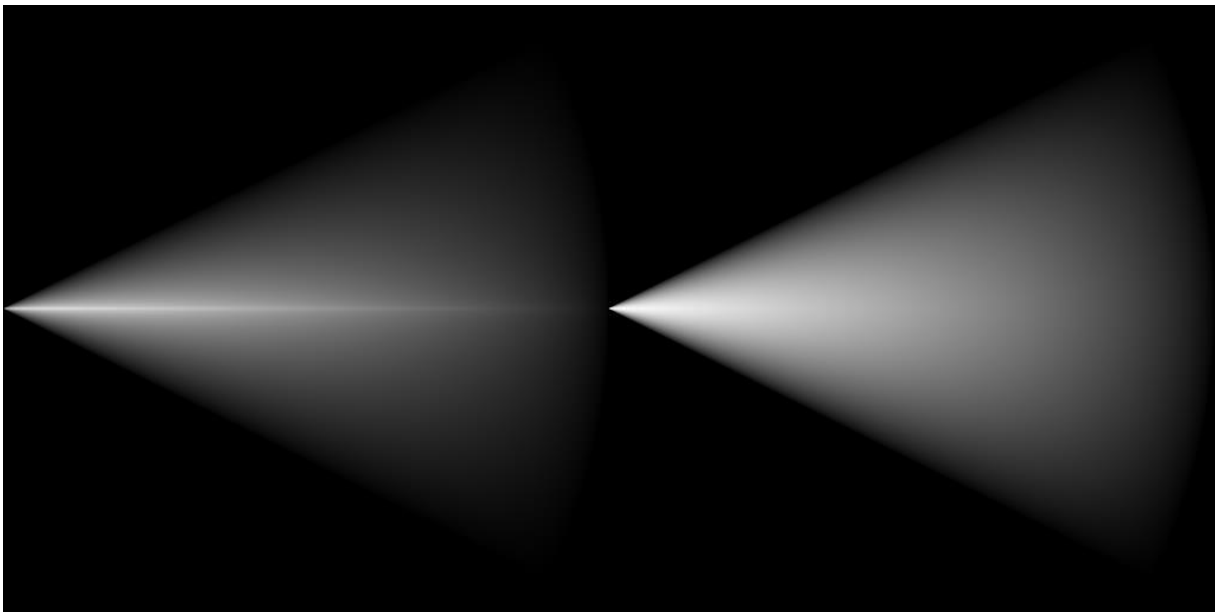
Joonis 2. Punktvalgus

Prožektorvalgus on valgusallikas, mis kiirgab valgust ühes kindlas suunas ja mille valgusvihu nurk ei ulatu üle 180 kraadi (näiteks taskulamp). Analoogselt punktvalgusele, peaks see olema automaatselt loodav kujutis ilma kasutajapoolsete lisanõueteta ning arvestama sarnaselt kaugushajuvust. Lisaks rakendub eksponentsiaalne hajuvus koonuse servade poole, millest on tingitud ka prožektorvalguse lisaomadus:

Tabel 2. Prožektorvalguse spetsiifilised omadused

Omadus	**Kirjeldus**	**Piirangud**
Koonuse valgusvihu hajuvus	Määrab ära valguskiirte tugevuse keskpunktist vaadatuna. Mida kõrgem hajuvus, seda tugevam on valgusvihk valguskoonuse servade pool.	Hajuvus ei tohi olla negatiivne.

Eelnevat omadust saab edukalt kirjeldada illustratsiooniga. Järgnevalt on toodud joonis, kus vasakul pool on madala ja paremal pool kõrge koonuse hajuvusega prožektorvalgustid:



Joonis 3. Madala hajuvusega prožektorvalgus

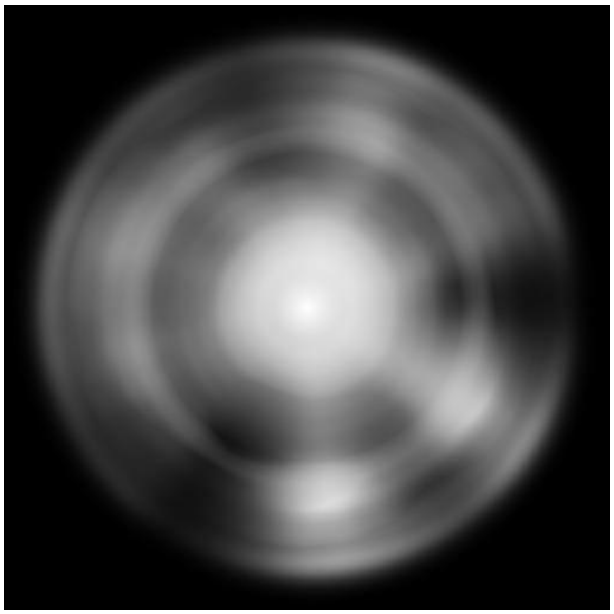
Joonis 4. Kõrge hajuvusega prožektorvalgus

Tekstuuritud valgus on valgusallikas, mille valgustugevuse määrab ära valgusallikale määratud tekstuur. See annab täieliku kontrolli kasutajale selle üle, millise kujuga on valgusvihk ja kui palju mingit valguse poolt hoomatavat pinnapunkti valgustatakse. Valguse hajuvust eraldi ei arvestata.

Kuna ei ole oluline, millise värvikomponendi järgi valgustugevust vaadeldakse, siis võetakse see rohelisest komponendist. See loob eelduse, et kasutaja poolt defineeritud tekstuur on must-valge ehk kõigi kolme komponendi väärtused on võrdsed.

Tabel 3. Tekstuuritud valguse spetsiifilised omadused

Omadus	**Kirjeldus**	**Piirangud**
Tekstuur	Tekstuur, mida kasutatakse valgustugevuse leidmiseks pikslil.	



Joonis 5. Tekstuuritud valgus [5]

Tekstuuri animeerimine võimaldab luua keerukamaid valgusefekte nagu näiteks küünlavalguse või lõkketule simuleerimine.

2.2 Varjud

Vari on regioon, kuhu valgusallikast jõudev valgus on takistatud läbipaistmatu objekti poolt.

Selleks, et varjude simuleerimine oleks üleüldse võimalik, peab teegi kasutajal olema võimalik määrata ära stseenis objektid, mis on valguse poolt läbimatud ehk teisisõnu „heidavad varje“ (reaalselt midagi ei heideta, vaid piiratakse valgusallika poolt valgustatust). Edaspidi kutsutakse selliseid objekte kestadeks.

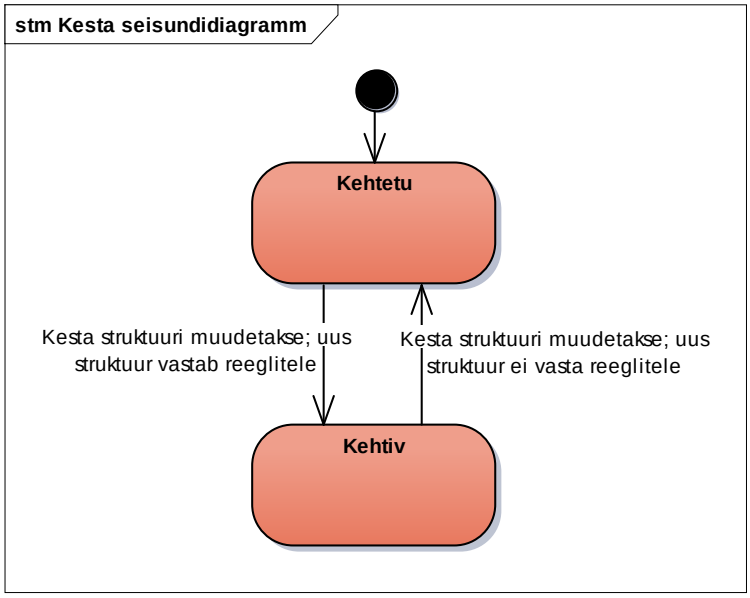
2.2.1 Kest

Kest on sisult tavaline hulknurk. Selleks, et kesta hulknurk oleks kehtiv - et teda oleks võimalik varjude simuleerimisel kasutada, peavad olema täidetud järgnevad reeglid:

- Kest peab koosnema vähemalt kolmest nurgast (vastasel juhul poleks tegu hulknurgaga)
- Kesta nurgad peavad moodustama lihtsa hulknurga (hulknurga, mille ükski külg ei ristu ühegi teise sama hulknurga küljega)

Kasutajal peab olema võimalik kesta struktuuri ehk hulknurga nurki defineerida ja muuta programmi käimisel. Selleks defineerib kest omale kaks seisundit: „kehtiv“ ja „kehtetu“. Kehtiv

kest on kest, mida arvestatakse varjude simuleerimisel; kehtetut hulknurka mitte. Seisundi muudatused toimuvad kesta hulknurga struktuuri muutmisel: kui uus struktuur vastab eeltoodud reeglitele, jäetakse või viiakse kest seisundisse „kehtiv“; kui ei vasta, siis seisundisse „kehtetu“.



Joonis 6. Kesta seisundidiagramm

Kest peab võimaldama nurki defineerida nii päripäeva kui ka vastupäeva. Süsteem peab suutma tuvastada, mis järjekorras nurgad on ette antud ning vajadusel pöörama järjekorra vastupidiseks. Käesolevas töös lepitakse kokku, et sisemiselt hakkab teek toimetama vastupäeva sätestatud hulknurkadega.

Üldjuhul soovitakse kesta nurki määrata mitte stseeni koordinaatruumis, vaid hulknurga enda lokaalses koordinaatruumis. Taoline lähenemine võimaldab erinevate kestade jaoks ära kasutada samu hulknurga struktuure. Selleks, et lokaalsest koordinaatruumist teisendada hulknurga koordinaadid stseeni koordinaatruumi, peab kest võimaldama muuta järgnevaid omadusi:

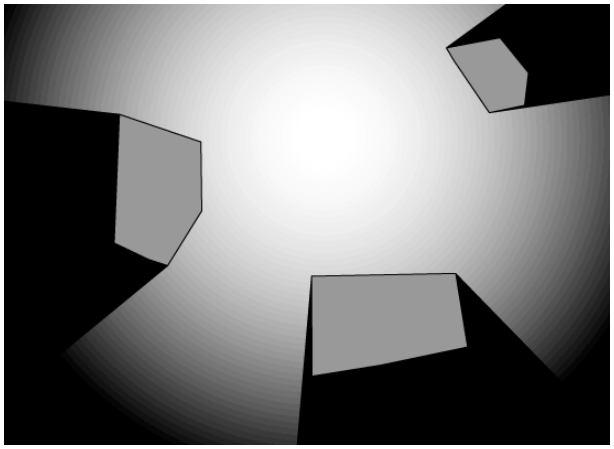
Tabel 4. Kesta omadused

Omadus	**Kirjeldus**	**Piirangud**
Asukoht	Määrab ära kesta asukoha simuleeritava maailma (stseeni) koordinaatruumis.	

Rotatsioon	Määrab ära, kuhu suunas kest on suunatud.	
Suurus	Määrab ära kesta suuruse.	Suurus ei saa olla negatiivne.

2.2.2 Täisvari (umbra)

Täisvari on piirkond, kuhu ühegi kohtvalgusallika valguskiired ei jõua. Kui valgusallika kese on vaadeldava stseeni suhtes piisavalt väike (raadius läheneb nullile), võib ette kujutada, et valguskiired väljuvad ainult valgusallika keskpunktist. Tõmmates sellest punktist sirge, mis läbib keha ääre punkte vaadeldava valgusallika suhtes, saadakse keha taha jääv täisvarjuline ala.

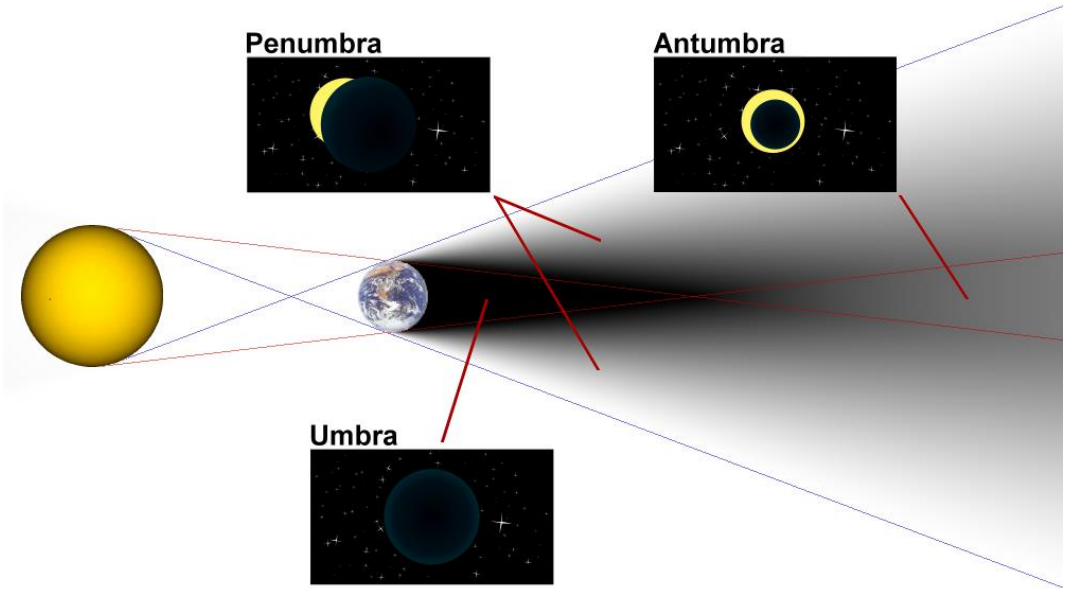


Joonis 7. Täisvari [6]

2.2.3 Poolvari (penumbra)

Kui valgusallikal on stseeni mõõtühiku suurustega võrreldes arvestatav raadius, ei välju valguskiired mitte enam ainult ühest punktist, vaid juba 2-mõõtmelisest ringist. Sellisel puhul määrab raadius ära tekkivate täis- ja poolvarjude geomeetrilised kujutised. Täisvarju regioon leitakse tõmmates kummastki valgusallika äärest (kesta suhtes) sirge, mis puutub kesta sama äärt. Poolvarju regioonid leitakse, kui läbi kummagi kesta ääre (valgusallika suhtes) tõmmatakse sirge, mis puutub kesta vastasäärt.

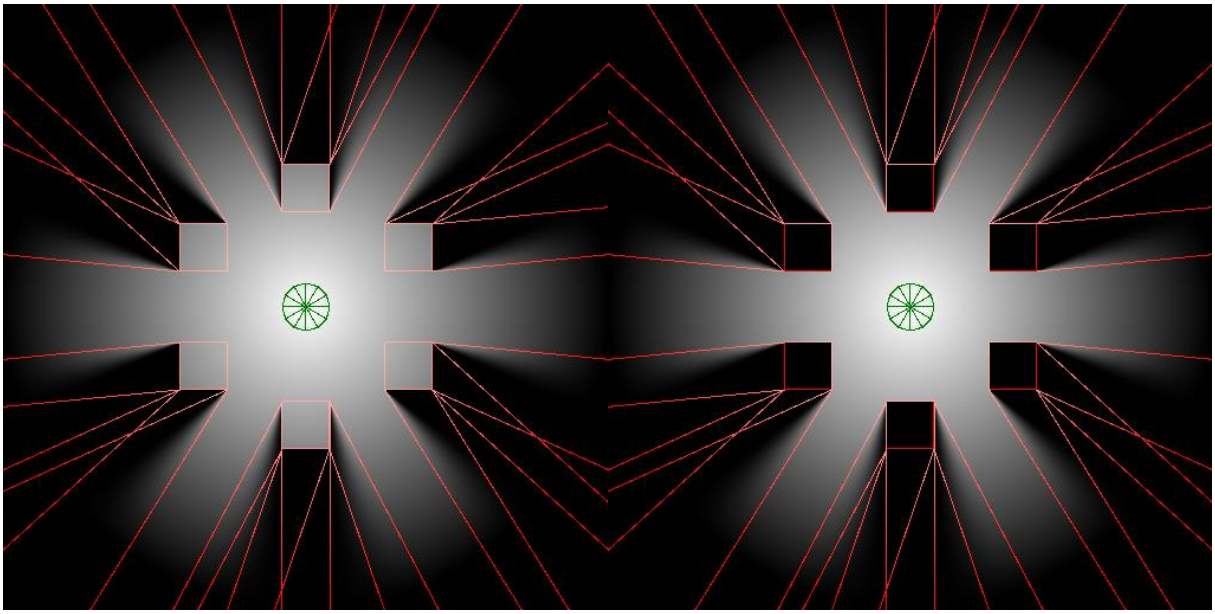
Peamiseks probleemiks jääb asjaolu, kuidas kujutada täisvarjult poolvarjule üleminekut selliselt, et see inimsilmale visuaalselt loomulik tunduks. Ei piisa ainult tooni vahetusest, vaid üleminek peaks olema sujuv, nagu see on illustreeritud järgneval joonisel:



Joonis 8. Poolvari [7]

2.2.4 Kestade valgustamine

Kuna tegeletakse 2-mõõtmelisel tasandil, siis taolist pinda võib 3-mõõtmelises maailmas ette kujutada erinevatel viisidel (näiteks otse ülevalt alla vaade või külje pealt vaade). Kuna kolmandat mõõdet ei arvestata, siis tekib küsimus, kas kesta ennast peaks valgustama või mitte. Lihtsuse huvides (teegi kasutajamugavust arvestades) ei tooda kolmandat dimensiooni kõrguse või sügavuse näol sisse, vaid probleem lahendatakse valgusallika omadusega, mis määrab ära, kas valgusallikas valgustab kestasid või mitte.



Joonis 9. Valgusallikas valgustab kestasid

Joonis 10. Valgusallikas ei valgusta kestasid

3. Realisatsioon

Käesolevas osas tegeletakse eeltoodud funktsionaalsuse teostamisega raamistikule *MonoGame* kasutades *C# 6* ja *HLSL* süntakseid. Teek suunatakse töötama *Windowsi* töölaua rakendusel *.NET 4.0* raamistikul *DirectX 11* graafikaplatvormil.

3.1 Nõuded

3.1.1 Riistvaralised nõuded

MonoGame'i rakenduse riistvaralised nõuded sõltuvad peaasjalikult igast rakendusest endast. Nõue, mida eeldatakse igalt rakenduselt on *DirectX 9.0c'd* toetav graafika protsessor. [8]

3.1.2 Tarkvaralised nõuded

Tarkvaralised nõuded valgusefektide teegi lähtekoodi kompileerimiseks ja jooksutamiseks:

- *DirectX End-User runtimes (June 2010)*
- *.NET 4.0+* raamistik
- *MonoGame 3.4+*
- *Visual Studio 2015+*

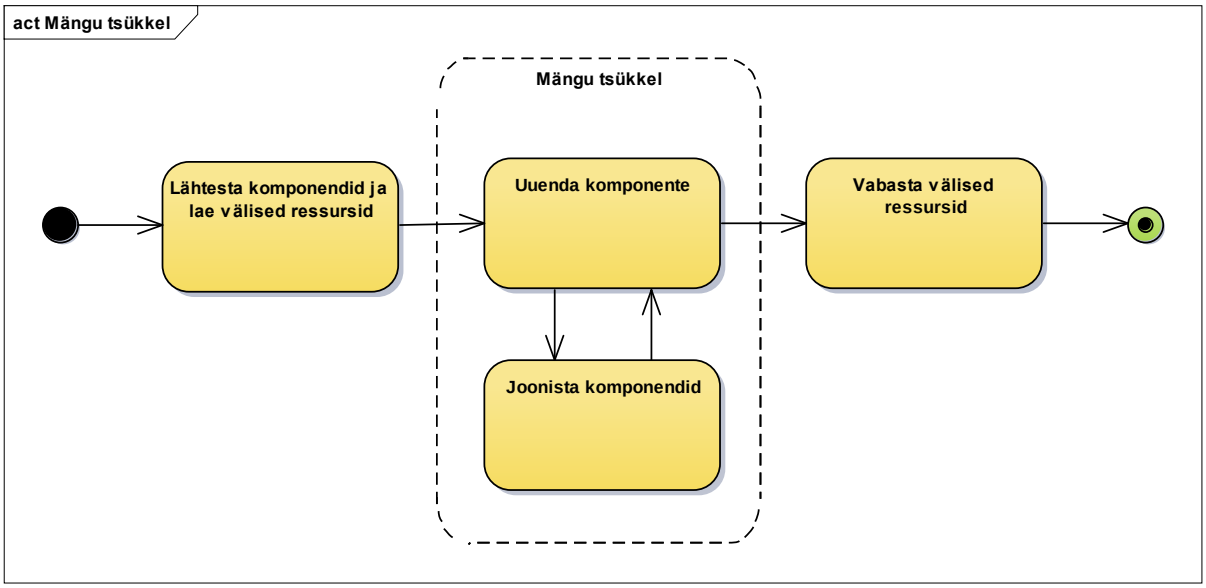
3.2 *MonoGame* rakendus

MonoGame raamistik on avatud lähtekoodiga ümberehitus *Microsoft XNA 4* raamistikust. Selle eesmärgiks on võimaldada *XNA* arendajatel *XBOX 360*, *Windowsi* ja *Windows Phone*'i mänge üle teisendada *iOSi*, *Androidi*, *Mac OS X*'i, *Linux*'i ja *Windows 8 Metro* platvormidele. *PlayStation Mobile*, *Raspberry PI* ja *PlayStation 4* platvormide tugi on töö kirjutamise hetkel arendamisel. [9]

Microsoft XNA on vabavaraline *.NET* raamistikul põhinev komplekt tööriistu ja teeke, et toetada videomängude arendust ja haldamist. *XNA* abstraheerib *Direct3D* programmeerimisliidest ning on suunatud peamiselt *indie* mängude arendajatele. Paraku lõpetati *XNA* edasiarendus 31. jaanuaril 2013. [10]

Oluline on teada, et *MonoGame*'i poolt kasutatav graafikaliides *Direct3D* käsitleb kogu rakenduses joonistatut läbi 3-mõõtmelise visualiseerimise. Isegi kui kasutaja jaoks on tegu 2-mõõtmelise stseeniga, toimub stseeni joonistamine analoogselt 3-mõõtmelisele stseenile. Üldjuhul saavutatakse 2-mõõtmelisuse kujutamine määrates y- või z-telje koordinaadid nulliks ning korrutades koordinaadid vastava lineaarteisenduse maatriksiga.

3.2.1 Mängu tsükkel



Joonis 11. Mängu tsükkel

MonoGame rakenduse töövoogu võib suures plaanis ette kujutada järgnevate protsessidena:

1. Esmalt rakendus lähtestatakse. Selles protsessis seadistatakse töövalmis rakendusele vajalikud komponendid ning laetakse välised ressursid nagu näiteks pildi- ja helifailid.

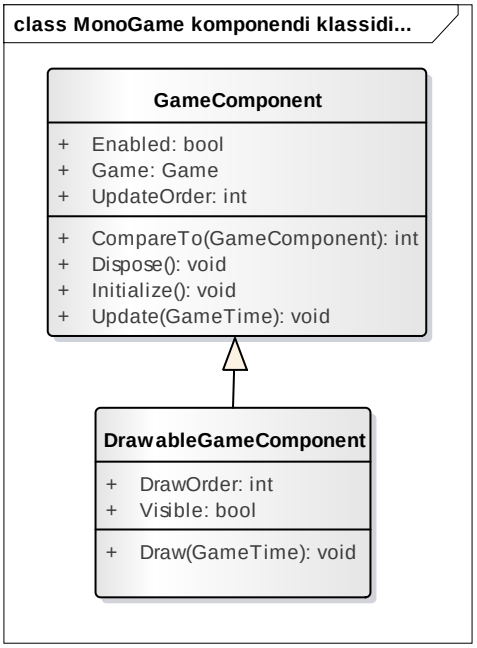
2. Seejärel algab mängu tsükkel, mis olenevalt seadistusele, tüüpiliselt uuendab rakenduse komponentide loogikat ja seejärel joonistab need rakenduse aknasse 30 või 60 korda sekundis. Igat taolist tsüklit nimetatakse kaadriks (*frame*). Vastavalt vajadusele, kui rakendus tuvastab, et eelmisest kaadrist on möödunud liiga palju aega ehk süsteem ei ole võimeline piisavalt kiiresti uuendama ja joonistama, võib rakendus otsustada kaadri vältel joonistamise vahele jätta. Loogika uuendamine on alati joonistamisest prioriteetsem.

3. Rakenduse sulgemisel omandatud ressursid vabastatakse ning rakendus suletakse.

3.2.2 Komponendid

Eesmärgiga kapseldada rakenduse loogiliselt kokkusobivaid osi, pakub *MonoGame* raamistik mängu komponentideks jaotamise kontsepti. Komponendid jagunevad suures plaanis kaheks: *GameComponent*, mis võimaldab ainult loogikat kapseldada ning *DrawableGameComponent*, mis lisaks loogikale pakub ka joonistamise võimalust. Uute võimaluste lisamiseks loob kasutaja uue klassi laiendades vastavalt vajadusele kumbagi eelnimetatud komponendi baasklassidest. Registreerides komponendi *MonoGame* mänguga, hakkab tema lähtestamist ja mängu tsüklist osavõtmist haldama raamistik ise.

Komponentideks, mille puhul on vajalik ainult loogika uuendamine, sobivad rakenduse iseseisvad loogilised osad nagu näiteks audio komponent muusika ja heliefektide haldamiseks või füüsika komponent mängumaailma füüsika simuleerimiseks. Joonistamisega seotud komponendiks võib näitena tuua eriefektide komponendi, mille ülesandeks on eriefektide nagu vihm, tuli, suits simuleerimine.

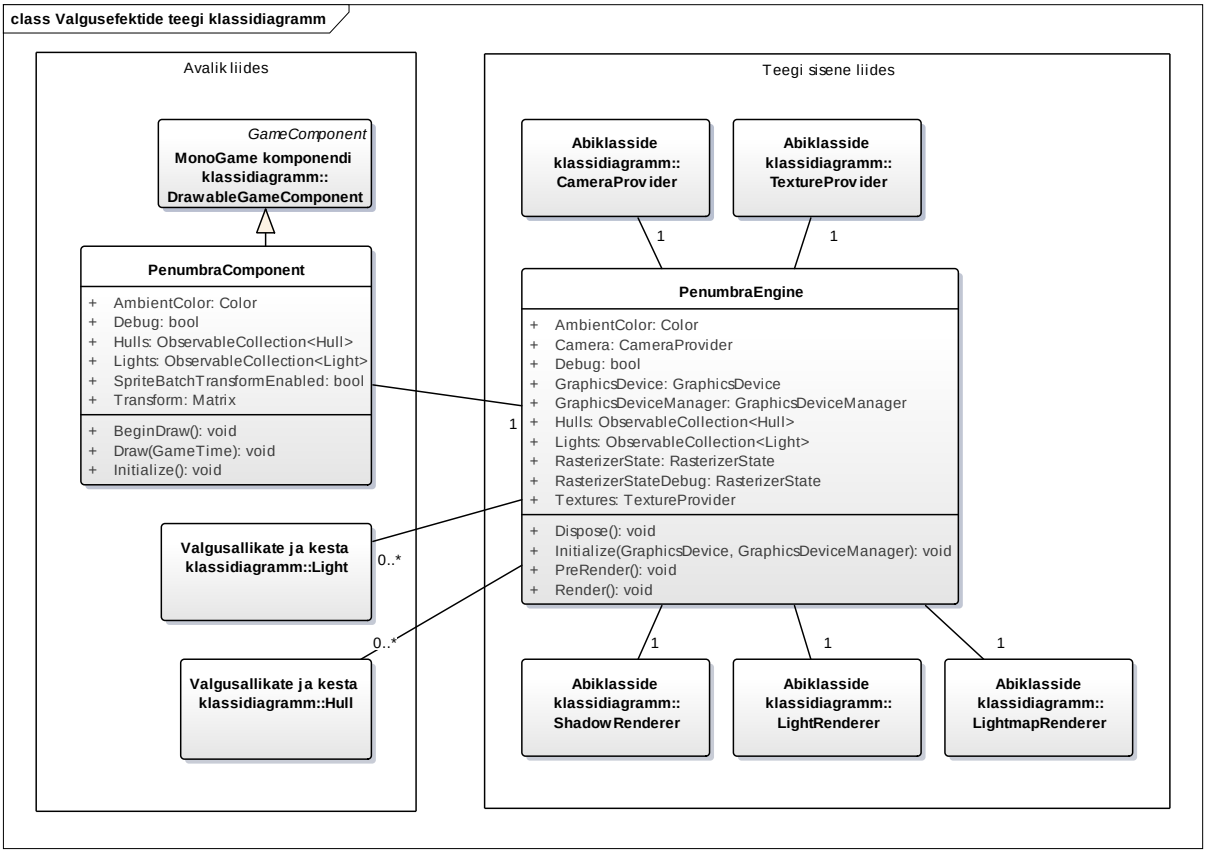


Joonis 12. MonoGame komponendi füüsilise disaini klassidiagramm

3.3 Valgusefektide teek

Antud teegi realisatsioonis loodaksegi uus taaskasutatav (konkreetsest rakendusest sõltumatu) joonistatav komponent, mille ülesandeks on eelnevalt projekteeritud valgusefektide funktsionaalsuse (vt peatükk 2) võimaldamine.

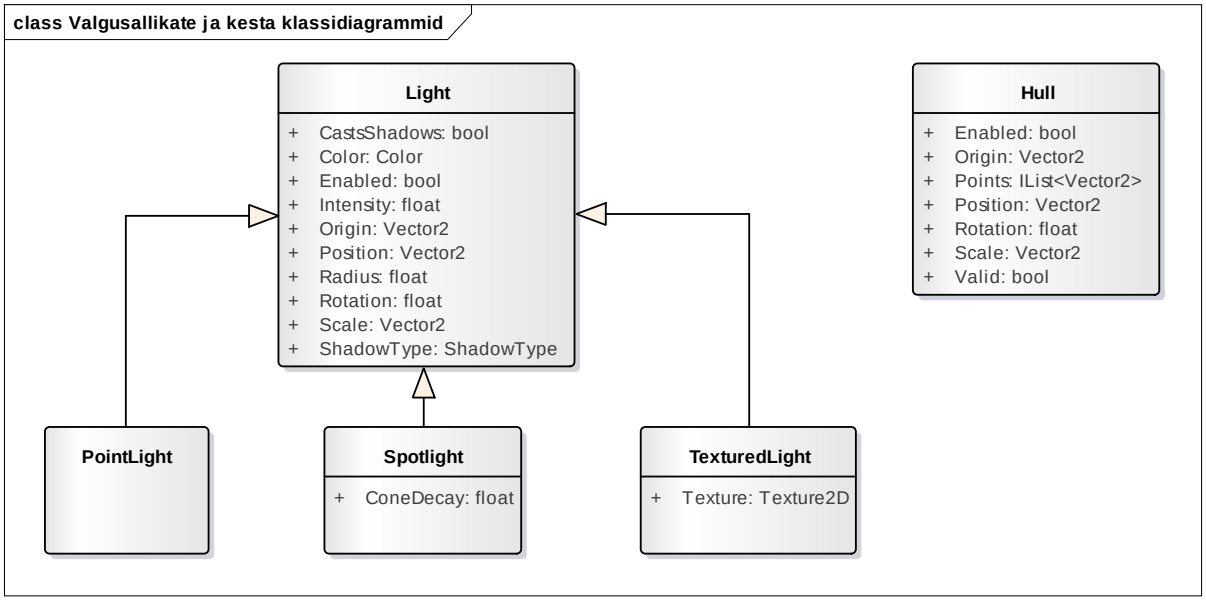
3.3.1 Füüsiline disain



Joonis 13. Valgusefektide teegi füüsilise disaini klassidiagramm

Eelneval joonises tuuakse suures plaanis eraldi välja teegi avalik ja mitteavalik liides. Avaliku liidese tuumklassiks on valgusefektide komponent (*PenumbraComponent*), mis pakub teegiga (*PenumbraEngine*) suhtlemiseks olulist funktsionaalsust läbi *MonoGame* rakendustele tuttava komponentliidese. Idee komponendi ja mootori eraldamises seisneb selles, et mootoril endal oleks võimalikult vähe sõltuvusi otse *MonoGame* raamistikule, võimaldades potentsiaalselt kergemat üleviimist mõnele teisele graafika raamistikule.

Lisaks komponendile endale, mängivad avalikus liideses olulist rolli projekteeritud eskiismudeli objektid nagu erinevat tüüpi valgusallikad ja kest.

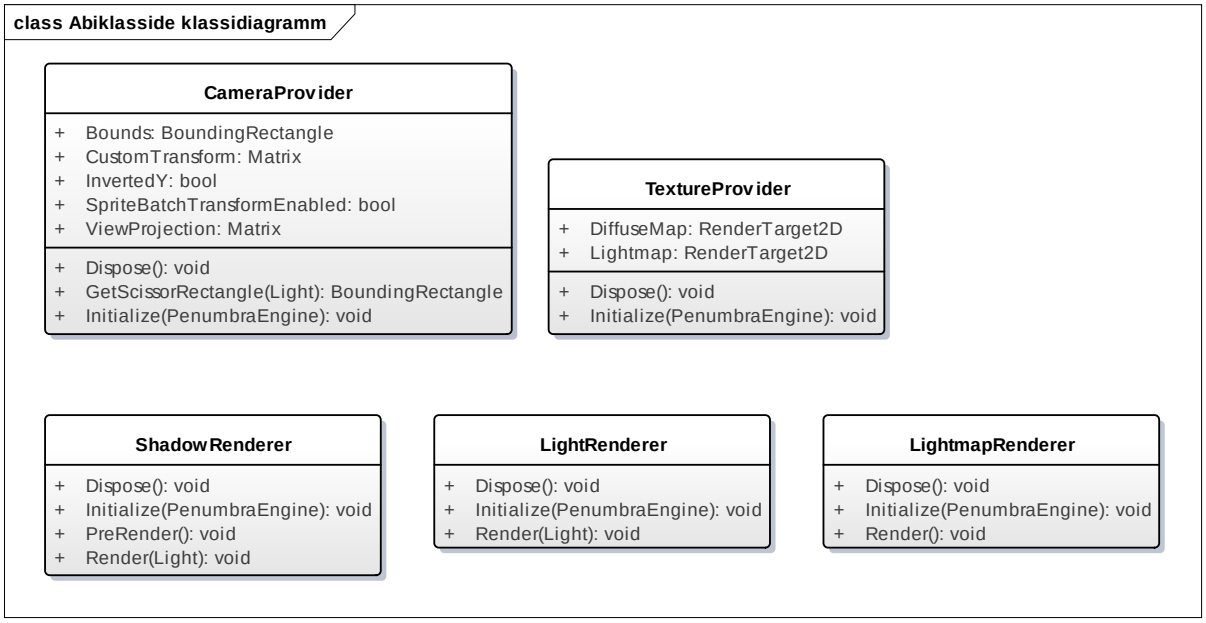


Joonis 14. Valgusallikate ja kesta füüsilise disaini klassidiagramm

Teegi siseses liideses on teegi funktsionaalsuse põhjal loogilisteks abiklassideks jaotatud varustajad (*providers*) ja visualiseerijad (*renderers*).

Varustajate eesmärk on võimaldada ülejäänud teegi osadele vajalikke ressursse. Kaamera varustaja hoolitseb teisendusmaatriksi arvutamise, *scissor rectangle'*i loomise ja ala leidmise eest, mis jääb ekraani vaatesse. Tekstuuri varustaja haldab tekstuuriressursside loomist graafika protsessorile.

Visualiseerijad teostavad joonistamisoperatsioone, mis on vajalikud valgusefektide kuvamiseks ekraanile. Varjude visualiseerija märgib ära valgusallika poolt varju jäävad alad (vt peatükk 3.4), valguse visualiseerija konkreetse valgusallika joonistamist (vt peatükk 3.5) ning valguskaardi visualiseerija sujutab teegi poolt loodud valguskaardi tagasi kasutaja poolt joonistatud *render targetile* (vt peatükk 3.6).



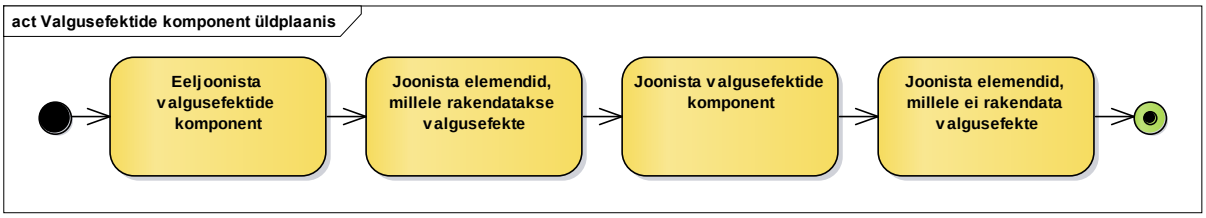
Joonis 15. Abiklasside füüsilise disaini klassidiagramm

3.3.2 Komponendi töövoog üldiselt

Valgusefektide komponent on kogu realiseeritava teegi tuum. Igasugune süsteemi konfiguratsioon, kaasa arvatud valgusallikate ja kestade lisamine/eemaldamine käib selle avaliku liidese kaudu.

Kuna tegu on puht visuaalse komponendiga, on *MonoGame* rakendusel ainult tarvilik seda joonistada, kuid mitte uuendada. Kuigi võib väita, et joonistamise ajal ka uuendatakse kestade ja valgusallikate andmestruktuure, siis need uuendused on vajalikud ainult joonistamise tarvis ning seetõttu teostatakse samuti joonistustsüklis.

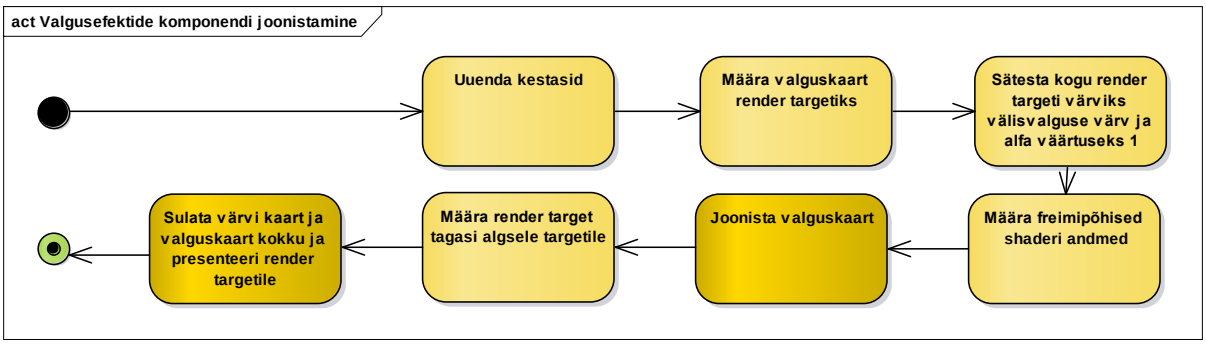
Paraku ainult joonistustsüklist komponendi funktsioneerimiseks ei piisa. Kuna korrektseks tööks on vajalik ka kasutaja poolt joonistatud värvi kaart ehk tekstuur, kuhu on joonistatud stseeni elemendid, millele soovitakse valgusefekte rakendada, siis lisaks on vaja teostada enne komponendi joonistamist ka eeljoonistamine. Eeljoonistamise eesmärk on talletada graafika konveierile sätestatud *render target* ning määrata sinna uus värvi kaart, mida kasutatakse hiljem valguskaardiga kokku sujutamisel. Oluline on, et kaadri vältel on eeljoonistusetapp läbitud enne kasutajapoolset joonistamist, mis omakorda on läbitud enne komponendi joonistamist. Kui kasutaja soovib joonistada ekraanile elemente, mida ei tohiks üldse valgustamisel arvestada (näiteks kasutajaliidese elemendid), siis peaks need joonistama peale valgusefektide komponendi joonistamist.



Joonis 16. Valgusefektide komponent üldplaanis

Järgnevalt tuuakse välja kirjeldused nii komponendi joonistamisest üldiselt kui ka täpsemalt joonistusprotsessi alamosade kohta. Üldjuhul esitatakse kõrgtasemeline vaade tegevusdiagrammina ning seejärel antakse detailsem seletus. Tumedama taustavärviga markeeritud tegevused kirjeldatakse järgnevates peatükkides täpsemalt lahti.

3.3.3 Komponendi joonistamise töövoog



Joonis 17. Valgusefektide komponendi joonistamine

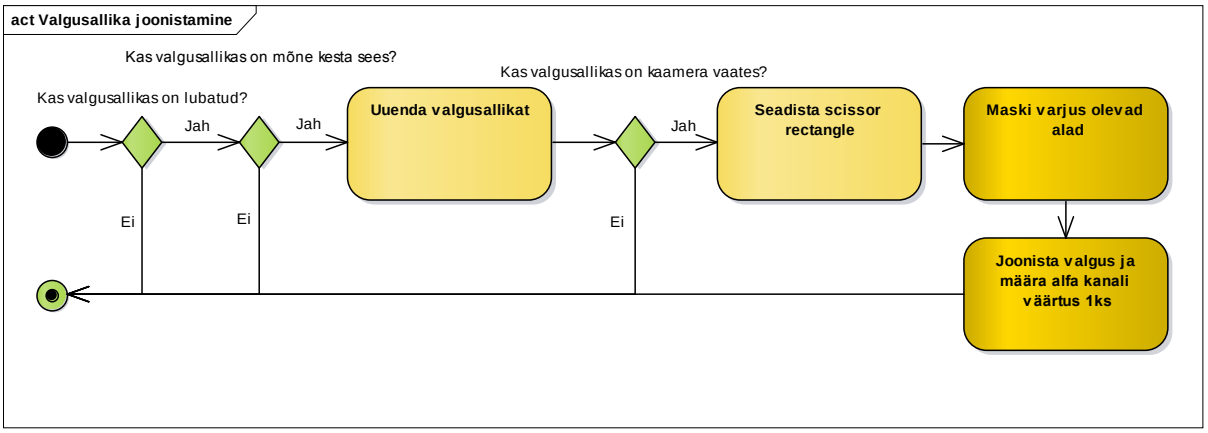
Esmase tegevusena uuendatakse kestade sisemisi andmestruktuure. Kestade omaduste muutmisel kasutaja poolt märgitakse kest „räpaseks„. Kesta uuendamisel kontrollitakse „räpasust“ ja vajadusel arvutatakse talle vajalikud andmed uuesti. Täpsemalt toimub kesta nurkade teisendamine kesta lokaalkoordinaatidest stseeni koordinaatsüsteemi.

Kuna *Direct3D* graafikakonveieri *render target* sätestatakse ümber, tuleb seal olemasolev *render target* hoiustada, et see peale komponendi joonistamist taastada. Eesmärk on teeki kasutava arendaja töövoogu võimalikult vähe häirida. Seejärel seadistatakse konveierile *render targetiks* valguskaart, mille peale kogu valgusefektide joonistamine hakkab toimuma.

Enne valguskaardile joonistamist on vaja tema andmed algväärtustada. Kuna andmeid hoitakse RGBA formaadis, kirjutatakse RGB kanalitesse komponendi välisvalguse värv ning alfa kanal A väärtustatakse ühtedega. Alfa kanalit hakatakse kasutama selleks, et iga kohtvalgusallika kohta määrata, kui palju igat pikslit valgusallikas valgustab. Alfa väärtus üks tähendab täielikult valgustatud pikslit; väärtus null täielikult valgustamata pikslit.

Järgmise tegevusena määratakse varjude maskimiseks vajalikud kaadripõhised *shaderi* andmed. Seejärel käiakse läbi kõik komponendile määratud kohtvalgusallikad ning joonistatakse need. Viimaks suunatakse konveieri *render target* tagasi algselt hoiustatud *render targetile* ning sujutatakse see kokku loodud valguskaardiga (vt peatükk 3.6).

3.3.4 Valgusallika joonistamise töövoog



Joonis 18. Valgusallika joonistamine

Esmalt teostatakse rida teste, et kindlaks teha, kas valgusallikat tasub üldse joonistada. Kui valgusallikas pole kasutaja poolt lubatud või valgusallika kese pesitseb mõne kesta sees, katkestatakse igasugune edasine toiming sellega. Et kindlaks teha, kas valgusallika poolt hõlmatav ala jääb kaamera vaatesse, tuleb valgusallika sisendstruktuure uuendada. Analoogselt kestade uuendamiselegi, kontrollitakse valgusallika „räpasust“ ning vajadusel arvutatakse hõlmatav ala stseeni koordinaatsüsteemis.

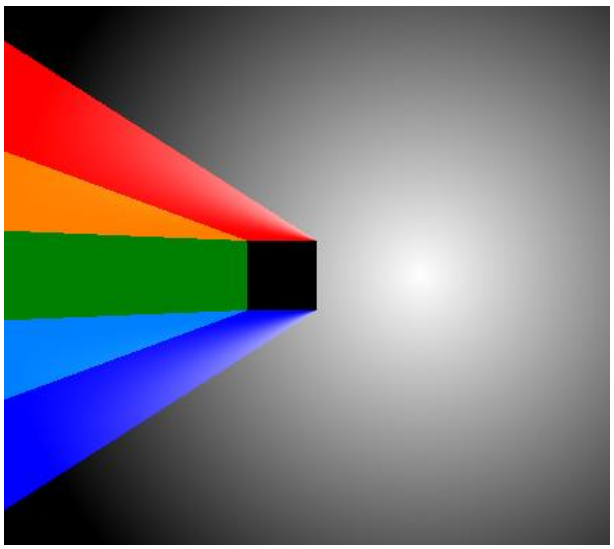
Kui eelnevad testid on edukalt läbitud, alustatakse valgusallika joonistamisega. Seadistatakse *scissor rectangle*, mis lubab piksleid joonistada vaid valgusallika poolt hõlmatavasse regiooni (väljapoole jäävad pikslid kärbitakse). Teostatakse varju jäävate regioonide osaline või täielik maskimine (vt peatükk 3.4), mis kujutab endast pikslite alfa väärtuse vähendamist, kus täisvarju puhul vähendatakse väärtus nulli, poolvarju puhul jääb alfa väärtus nulli ja ühe vahepeale. Viimaks joonistatakse valgusallikas (vt peatükk 3.5) ja taaslähtestatakse valguse alla jäävate pikslite alfa väärtus üheks. Taaslähtestamine on oluline, et ühe valguse varjude maskimine ei tekitaks visuaalseid artefakte järgmise valgusallika joonistamisel.

3.4 Varjude maskimine (shadow masking)

Maskimine toimub *render targeti* alfa kanali peal. Selle eesmärk on konkreetse valgusallika poolt hõlmatava ala puhul vähendada alfa väärtust piirkondades, mis jäävad käsitleva valgusallika suhtes varju. Varju geomeetriaks on nelinurgad, mis projekteeritakse kesta igast küljest. Projekteerimist teostatakse graafika protsessoril *vertex shadingu* etapis. Piksel *shaderis* arvutatakse, kui palju piksli alfa kanali väärtust vähendada.

3.4.1 Lähtestamine

Varjude maskimiseks on vajalik *vertex*- ja indekspuhvrite seadistamine vajaliku geomeetriaga. Kasutatakse dünaamilisi puhvreid, mis tähendab, et protsessoril on õigus puhvri andmeid muuta ka siis, kui puhver on juba laetud graafika protsessorile. Jõudluse parendamiseks hoiustatakse kõikide kestade andmed ühes ja samas *vertex*- ja indekspuhvris. Kokku on *vertex*- ja indekspuhverpaare kaks. Ühte paari hoiustatakse varjude jaoks vajaminev geomeetria, teise paari hoiustatakse kestade endi geomeetria. Teine paar on oluline, et parandada ebasobivast varjugeomeetriast ja maskimisest tekkinud vigu ning määrata ära, kas valgus valgustab kesta ennast või mitte. Kestade puhul määratakse *vertexiteks* kestade nurgad. Varjude puhul luuakse kesta iga külje kohta neli *vertexit*: kaks tükki jäetakse kesta külge tähistama, kaks tükki projekteeritakse eemale, kui tegu on valgusallika suhtes eemale suunatud külgedega. Iga *vertex* sisaldab infot külje mõlema nurga asukoha kohta ning abistruktuuri, mis tähistab ära, kas tegu on projekteeritava *vertexiga* ja kas *vertexit* tuleb projekteerida valgusallika vasakust või paremast äärest.



Joonis 19. Varju geomeetria projekteerimine illustreeritult

Joonisel kujutavad punane, roheline ja sinine värv kesta igast küljest valgusallika suhtes eemale projekteeritud varjude geomeetriat. Oranž ja helesinine tähistab ala, kus kahest küljest projekteeritud geomeetria kattub.

Seadistatakse nii varjude kui ka kestade joonistamiseks vajalikud ühtsed graafikakonveieri seisundid. Seisundid on välja toodud tabeli kujul ning seisundi parameetritest ainult olulised.

Tabel 5. Varjude joonistamise rasterdamisseisund

Nimetus	**Väärtus**
Kärpimisviis	Päripäeva või vastupäeva (olenevalt kasutaja poolt seadistatud teisendusmaatriksile)
***Scissor test* lubatud**	Jah
Täiteviis	Tahke

Tabel 6. Varjude joonistamise sügavus-šabloonseisund

Nimetus	**Väärtus**
Sügavuspuhver lubatud	Ei
Šabloontest lubatud	Ei

Tabel 7. Varjude joonistamise sujutamisseisund

Nimetus	**Väärtus**
Lubatud värvikanalid	Alfa
Alfa sujutamisfunktsioon	Inverteeritud lahutamine
Alfa allika kordaja	Üks
Alfa sihtmärgi kordaja	Üks

Tabel 8. Kestade joonistamise sujutamisseisund

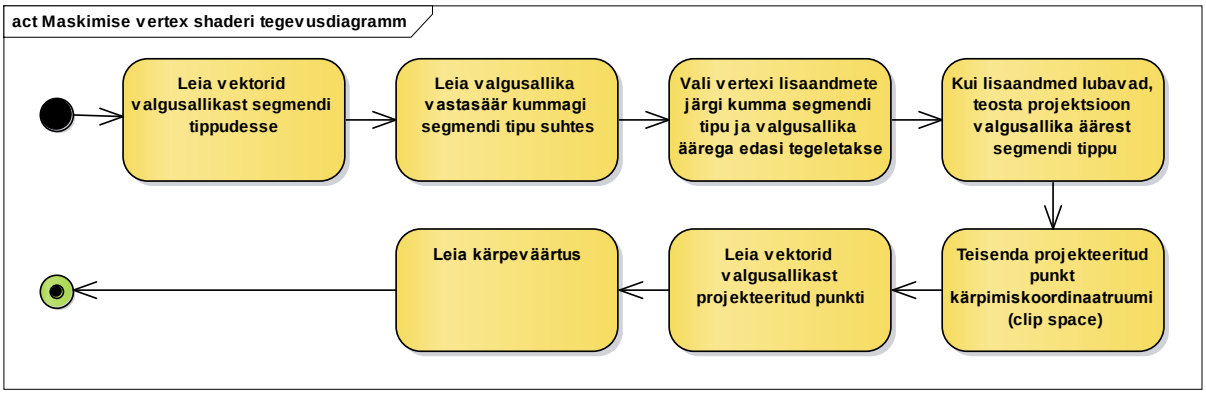
Nimetus	**Väärtus**
Lubatud värvikanalid	Alfa

Alfa sujutamisfunktsioon	Liitmine
Alfa allika kordaja	Üks
Alfa sihtmärgi kordaja	Null

Graafika protsessoril seadistatakse aktiivseteks *vertex*- ja indekspuhvriteks varjugeomeetria puhvrid. Laetakse *shaderile* vajalikud andmed: valgusallika positsioon ja raadius. Määratakse varjude joonistamiseks vajalikud *vertex*- ja piksel *shaderid*.

3.4.2 *Vertex shaderi* algoritm

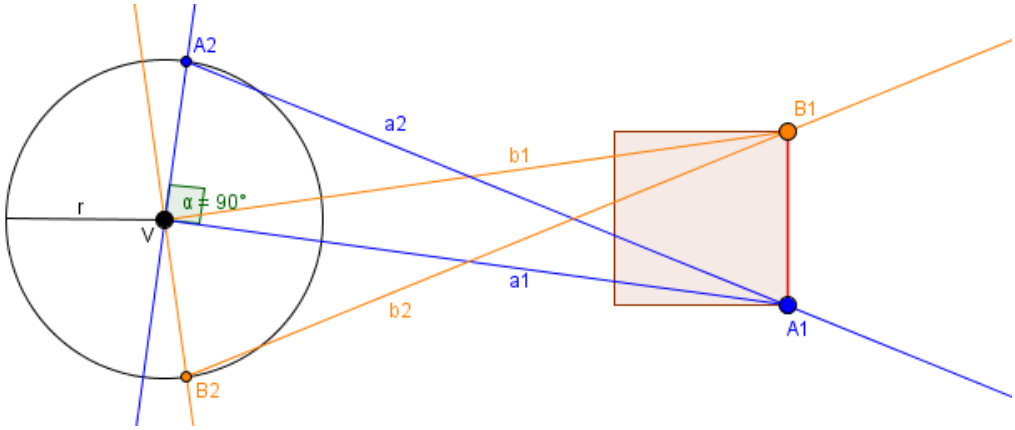
Vertex shaderil on kolm ülesannet: kindlaks teha, kas protsessitav kesta külg on valgusallika poolt eemale suunatud või mitte (et mitte tekitada varje valgusallika poole), vajadusel projekteerida külje segmendi punkt eemale ning arvutada välja vektorid, mis iseloomustavad jõudva valguse hulka sinna punkti.



Joonis 20. Maskimise vertex shaderi tegevusdiagramm

HLSL pakub samu voo kontrolli käsklusi, *if-else-then*, *for* ja *while*, nagu enamus programmeerimiskeeled. Paraku nende teostus erineb *CPU*'l põhinevatel käsklustest. *GPU* paralleelse iseloomu tõttu võib nende teadvustama kasutamine negatiivselt mõjuda rakenduse jõudlusele. Nende probleemide vältimiseks on teadlikult püütud vältida voo kontrolli käsklusi. Seetõttu võib kohati teostatavate operatsioonide järjekord tunduda ebaloogiline. Näiteks projekteeritakse ka neid külgi, mis on suunatud valgusallika poole, kuigi on teada, et nendest tekkiv varjugeomeetriat ei tohiks arvestada. Küll aga kärbitakse need hiljem piksel *shaderi* etapis. [11]

Projekteerimiseks leitakse vektorid a1 ja b1 valgusallikast V raadiusega r mõlemasse külje segmendi tippu A1 ja B1. Nende põhjal leitakse omakorda valgusallika ääre punktid A2, B2 nii, et valgusallika poolt vaadatuna vasaku segmendi tipu suhtes leitakse punkt valgusallika parempoolsel äärel ning parempoolse segmendi tipu suhtes valgusallika vasakpoolsel äärel. Tehakse *vertexi* lisaandmete põhjal otsus, kumma tipuga edasi tegeletakse ning kas seda tippu projekteeritakse eemale või mitte. Kui tippu projekteeritakse, siis leitakse siht a2 või b2 ning seadistatakse vektori w-komponent nulliks. See tagab selle, et perspektiivjagamise käigus jagatakse x- ja y-komponendid läbi väga väikese nullile läheneva arvuga, mida võib ette kujutada, kui vektori projekteerimist piki sihti lõpmata kaugele. See on sobilik, sest väljapoole valgusulatust jäävad pikslid kärbitakse *scissor test* etapil. Kuna ühe ja sama külje kohta käib läbi *vertex shaderi* neli *vertexit* (kummagi tipu kohta kaks), siis saadaksegi neli erinevat käitumismalli, mis tagavad selle, et peale nende nelja töötlemist on tulemuseks nelinurk, mis kujutab varju geomeetriat.



Joonis 21. Kesta külje tippude projekteerimine

```
// Leia vektorid a1 ja b1.
float2 toSegmentA = vertexInput.SegmentA - LightPosition;
float2 toSegmentB = vertexInput.SegmentB - LightPosition;

// Leia punktid A2 ja B2.
float2 toLightOffsetA = float2(-LightRadius, LightRadius)*normalize(toSegmentA).yx;
float2 toLightOffsetB = float2(LightRadius, -LightRadius)*normalize(toSegmentB).yx;
float2 lightOffsetA = LightPosition + toLightOffsetA; // 90 kraadi vastupäeva.
float2 lightOffsetB = LightPosition + toLightOffsetB; // 90 kraadi päripäeva.

// Vali, kas käsitletakse segmendi tippu A1 või B1 ja valgusallika äärepunkti A2 või B2.
float2 position = lerp(vertexInput.SegmentA, vertexInput.SegmentB, vertexInput.Stencil.x);
float2 projectionOffset = lerp(lightOffsetA, lightOffsetB, vertexInput.Stencil.x);

// Vajadusel projekteeri valitud segmendi tipp piki sihivektorit a2 või b2 lõpmata kaugele.
float4 projected = float4(position - projectionOffset*vertexInput.Stencil.y, 0.0, 1.0 -
vertexInput.Stencil.y);
```

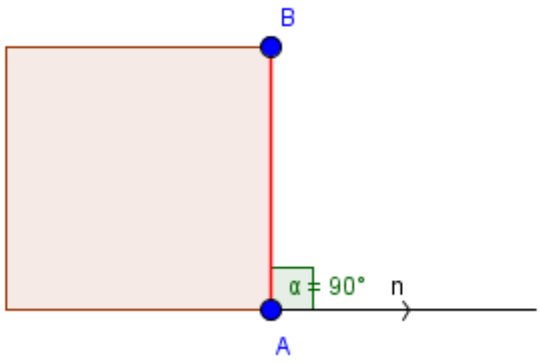
Toimub asukoha viimine stseeni koordinaatruumist kärpekoordinaatruumi (*clip space*) korrutades käsitletav segmendi punkt *projected* kasutaja poolt defineeritud teisendusmaatriksiga *ViewProjection* kasutades *HLSL mul* funktsiooni.

```
vertexOutput.Position = mul(projected, ViewProjection);
```

Järgmisena leitakse, kui palju valgust tuleb kummastki segmendi äärest. See on vajalik piksel *shaderi* etapis, et arvutada, kui palju käesolev piksel varju jääb ehk kui palju pikslit maskida. [12]

```
vertexOuptu.PenumbraA = mul(projected.xy - vin.SegmentA*projected.w,
Invert(float2x2(toLightOffsetA, toSegmentA)));
vertexOutput.PenumbraB = mul(projected.xy - vin.SegmentB*projected.w,
Invert(float2x2(toLightOffsetB, toSegmentB)));
```

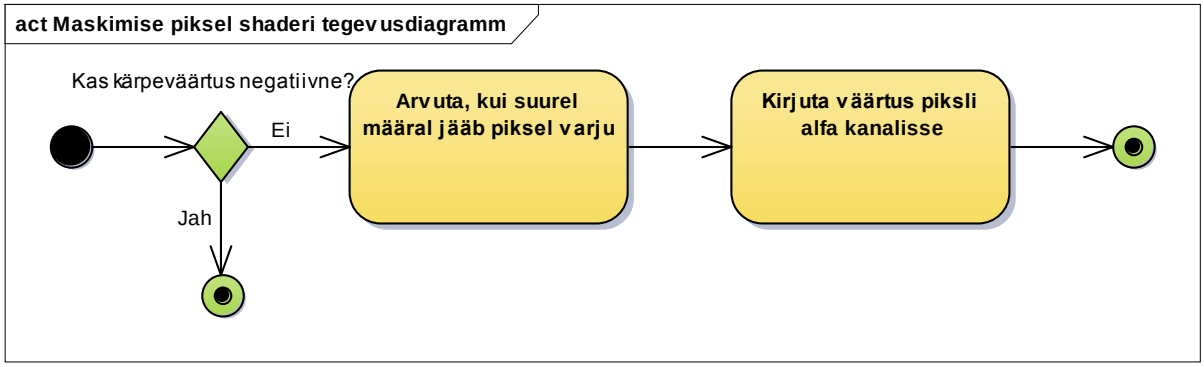
Viimasena leitakse kärpeväärtus. Selleks arvutatakse pinnanormaal külje suhtes. Kuna *vertexid* võimaldatakse vastupäeva, siis leitakse normaal, kui lahutada külje tipust B tipp A ja saadud vektorit n keerata 90 kraadi päripäeva. Sooritatakse pinnanormaali ja projekteeritud vektori vahel vektorite skalaarkorrutis. Skalaarkorrutise omadustest tulenevalt, kui korrutise väärtus on negatiivne, on pinnanormaal suunatud valguse suunas ning piksel *shaderi* etapis saab selle pikseli arvutuse vahele jätta (vastasel juhul joonistatakse varje valgusallika poole).



Joonis 22. Kesta külje pinnanormaali leidmine

```
// Leia segmendi pinnanormaal keerates vektorit A’st B’ni 90 kraadi päripäeva.
float2 clipNormal = (vertexInput.SegmentB - vertexInput.SegmentA).yx*float2(1.0, -1.0);
// Leia kärpeväärtus teostades skalaarkorrutis leitud normaali ja projekteeritud
punkti suhtes.
vertexOutput.ClipValue = dot(clipNormal, projected.xy – position*projected.w);
```

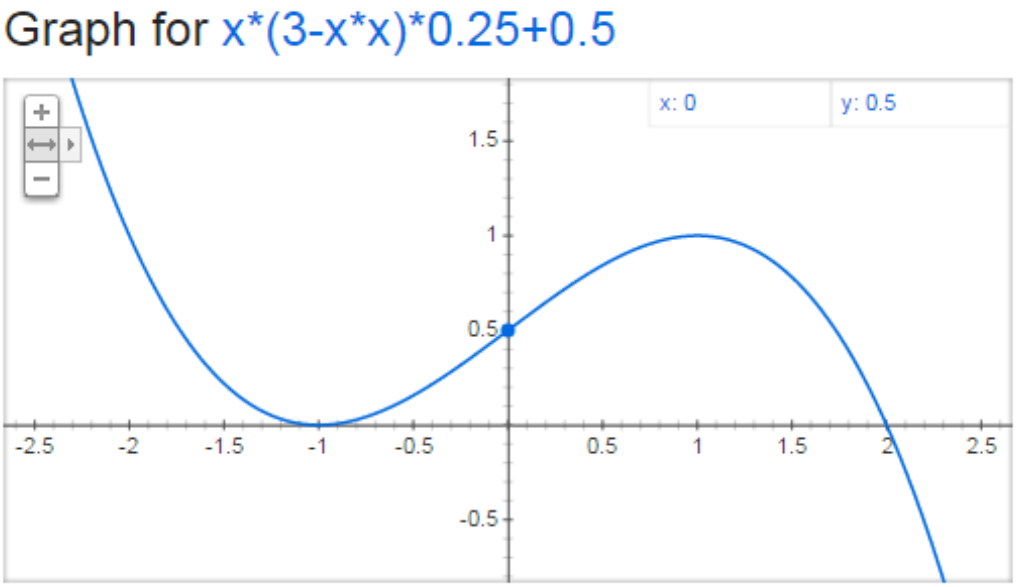
3.4.3 Piksel *shaderi* algoritm



Joonis 23. Maskimise piksel shaderi tegevusdiagramm

Piksel *shader* kärbib välja pikslid, mille kärpeväärtus on negatiivne, kasutades selleks *HLSL clip* funktsiooni. Kärpimata pikslite puhul leitakse mõlema poolvarju vektori x-, y-telje suhted ning arvutatakse kummagi põhjal maskimise väärtus. [12]

```
float FindOcclusionValue(float2 penumbra)
{
	float ratio = clamp(penumbra.x/penumbra.y, -1.0, 1.0);
	return ratio*(3.0 - ratio*ratio)*0.25 + 0.5;
}
```



Joonis 24. Valem maskimisväärtuse leidmiseks

Lõplik piksli väärtus väljastatakse liites mõlemast servast leitud maskimise väärtused ja lahutades sealt ühe maha, sest kummastki servast heidetud varjud on üksteist välistavad. Kuna tegeletakse ainult alfa kanaliga, siis väljastatava struktuuri RGB-komponentide väärtus ei mängi rolli ning seadistatakse nullideks.

```
float occlusionA = FindOcclusionValue(pixelInput.penumbraA);
float occlusionB = FindOcclusionValue(pixelInput.penumbraB);
return float4(0.0, 0.0, 0.0, occlusionA + occlusionB - 1);
```

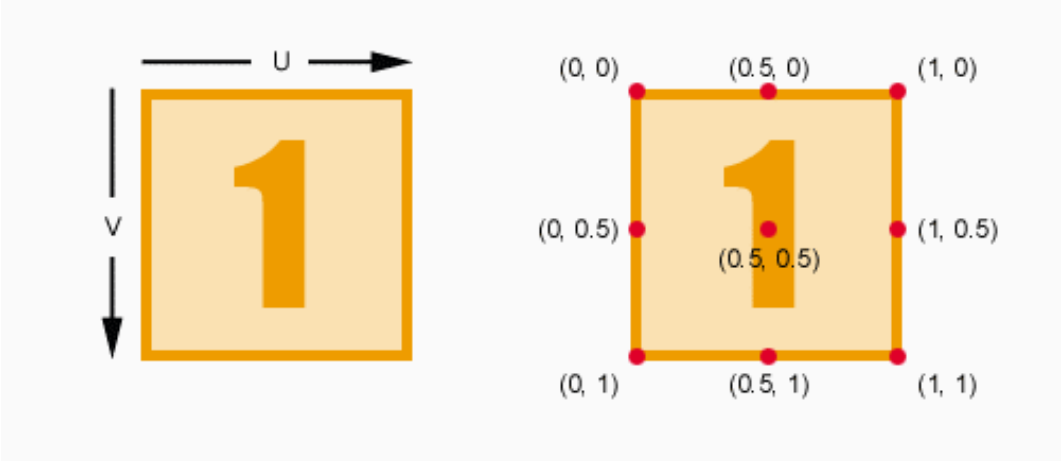
3.5 Valgusallika joonistamine

Valgusallika joonistamisel joonistatakse vastavalt valgusallika mõõtmetest tulenev ristkülik. Teades valgusallika keset ja olenevalt valgusallika tüübist, leitakse ristküliku igale pikslile vastav valgustugevus ning see värvitakse valguse värviga. Arvestatakse maskimisetapil kirjutatud alfaväärtust ehk varjus asetsevale pikslile kas ei rakendata üldse valgust või rakendatakse seda väiksemal määral (poolvarju korral).

3.5.1 Lähtestamine

Kõigi valgusallikate puhul kasutatakse ühte staatilist *vertex puhvrit*. Kuna joonistatakse topoloogiaga *triangle strip*, siis indekspuhver pole vajalik. Valgusallika *vertexi* tüübi puhul on oluline hoida seal lisaks asukohale ka tekstuuri koordinaate. Oluline on, et ristküliku küljed oleksid ühikpikkusega. See võimaldab valguse suurusega läbi korrutades koheselt saada vajalike mõõtmetega ristküliku.

Tekstuuri koordinaate kasutatakse, et märkida punkt tekstuuril. Kuna tekstuur on 2-mõõtmeline, siis on sellise punkti märkimiseks vaja kahte väärtust: U ja V. Kui (0, 0) märgib ära tekstuuri ülemise vasaku nurga, siis U väärtus määrab ära mitu ühikut tuleb alla liikuda ja V määrab ära mitu ühikut paremal liikuda selleks, et leida punkt tekstuuril. Antud teegi puhul peavad U, V väärtused jääma vahemikku 0 kuni 1.



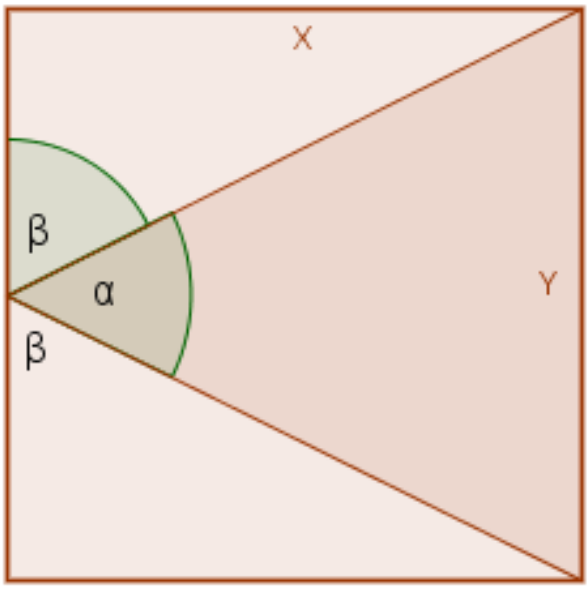
Joonis 25. UV kaardistamine [13]

Graafikakonveieri seisundid sarnanevad suures joones maskimisetapil kasutatuga: sügavus-šabloontest on välja lülitatud ning kasutatakse sama rasterdamisseisundit. Sujutamisseisundil on tähtis järgnev seadistus:

Tabel 9. Valgusallika joonistamise sujutamisseisund

Nimetus	**Väärtus**
Lubatud värvikanalid	Kõik
Värvi sujutamisfunktsioon	Liitmine
Värvi allika kordaja	Sihtmärgi alfa
Värvi sihtmärgi kordaja	Üks
Alfa sujutamisfunktsioon	Liitmine
Alfa allika kordaja	Üks
Alfa sihtmärgi kordaja	Null

Tekstuuritud ja prožektorvalguse puhul on oluline ka neile vastavate parameetrite uuendamine *shaderites*. Tekstuuritud valgusel sätestatakse valguse tekstuur. Prožektorvalguse puhul on vajalik seadistada tema prožektorist tuleneva valgusvihu koonuse nurk. Valgusallika kese pesitseb tema ristküliku vasaku külje keskpunktis. Koonuse nurk α leitakse ristküliku külgede pikkuste järgi:



Joonis 26. Prožektorvalguse koonuse nurga leidmine

$$\alpha = \pi - 2\beta = \pi - 2\arctan(\frac{2x}{y})$$

3.5.2 *Vertex shaderi* algoritm

Vertex shaderi eesmärk on teisendada valgusallika ristküliku nurgad ristküliku lokaalkoordinaatruumist kärpekoordinaatruumi. Selleks, et perspektiivjagamise tulemusena saadaks korrektsed asukohad normaliseeritud seadme koordinaatruumis, sätestatakse korrutatava vektori w-komponendiks 1. Tekstuuri koordinaadid saadetakse edasi ilma lisatoiminguteta.

```
vertexOutput.Position = mul(float4(vertexInput.Position.x, vertexInput.Position.y, 0.0,
1.0), WorldViewProjection);
vertexOutput.TextureCoordinate = vertexInput.TextureCoordinate;
```

3.5.3 Piksel *shaderi* algoritm

Iga valgusallika tüübi puhul kasutatakse erinevat *shaderit*. *Shaderite* ülesanne on valgustugevuse arvutamine konkreetse piksli korral (v.a tekstuuritud valgus) ja valguse värvi väljastamine pikslile.

Punktvalgusallika kese asub joonistatud ristküliku keskel. Kuna valgusallika keskpunktis on tekstuuri koordinaadi väärtus (0.5, 0.5), siis leitakse piksli kaugus *halfMagnitude* valgusallika keskpunktist ning valgustugevus *alpha* lineaarselt vastavalt kaugusele keskpunktist nii, et

keskpunktis on valgustugevus üks, ääres null. Negatiivsed väärtused piiratakse nulli *HLSL saturate* funktsiooni abil.

```
float halfMagnitude = length(pixelInput.TextureCoordinate - float2(0.5, 0.5));
float alpha = saturate(1.0 - halfMagnitude * 2.0);
```

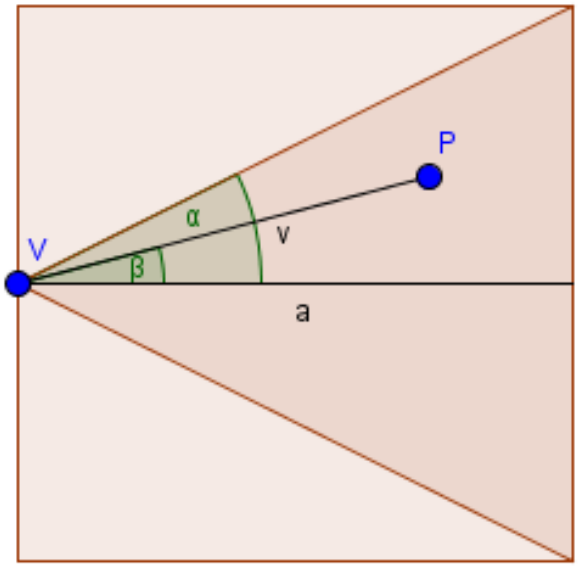
Lõplik *shaderi* poolt väljastatav arvutatakse valgusallika värvi RGB komponentide läbikorrutamisel leitud väärtusega *alpha*. Lisaks võetakse arvesse valgusallika intensiivsuse, millega teostatakse valgusvärvi astmesse tõstmine. Alfa kanalisse väljastatakse alati väärtus 1, et nullida ära maskimisest kirjutatud väärtus järgmise valgusallika jaoks.

```
float3 lightColor = LightColor * alpha;
lightColor = pow(lightColor, LightIntensity);
return float4(lightColor, 1.0);
```

Nagu eelpool sai välja toodud, siis prožektorvalguse puhul arvestatakse valgustugevust tema ristküliku vasaku külje keskpunktist. Piksel *shader* eeldab lisaks *vertexi* andmetele ka koonuse nurka ja hajuvusfaktorit. Alustuseks leitakse järjekordselt *vertexi* tekstuuri koordinaatide põhjal valgusvektor, tema pikkus ning normaliseeritud (pikkusega üks) sihivektor.

```
float2 lightVector = (pixelInput.TextureCoordinate - float2(0.0, 0.5));
float magnitude = length(lightVector);
float2 lightDirection = lightVector / magnitude;
```

Järgnevalt leitakse nurk leitud sihivektori ja prožektori suuna vahel. Kui leitud nurk on suurem kui kogu koonuse nurk ehk valgustatav punkt jääb koonusest väljapoole, seatakse valgustugevuskordaja *occlusion* nulliks *HLSL step* funktsiooni abil.



Joonis 27. Prožektorvalguse valgusvektori nurga leidmine ja võrdlemine koonuse nurgaga

```
float halfAngle = acos(dot(lightDirection, float2(1.0, 0.0)));
float occlusion = step(halfAngle, ConeHalfAngle);
```

Lõpliku valgustugevuse *alpha* leidmiseks on vajalik arvestada kaugushajuvust, koonushajuvust ja leitud *occlusion* väärtust. Kui kaugushajuvus hajutab valgustugevust, mida kaugemal valgusallikast punktiga on tegemist, siis koonushajuvus hajutab valgustugevust, mida koonuse ääre poole oleva punktiga on tegemist. Koonushajuvuse arvestamisel võetakse hajuvusfaktorit arvesse.

```
float distanceAttenuation = saturate(1.0 - magnitude);
float coneAttenuation = 1.0 - pow(halfAngle / ConeHalfAngle, ConeDecay);
float alpha = distanceAttenuation * coneAttenuation * occlusion;
```

Väljastatava värvi arvutamine on analoogne punktvalgusallikaga.

Tekstuuritud valgusallika puhul on *shaderi* protsess kõige lihtsam. Valgustugevus *alpha* võetakse kasutaja poolt etteantud tekstuurilt otse tekstuuri koordinaatide põhjal.

```
float alpha = Texture.Sample(TextureSampler, pixelInput.TextureCoordinate).x;
```

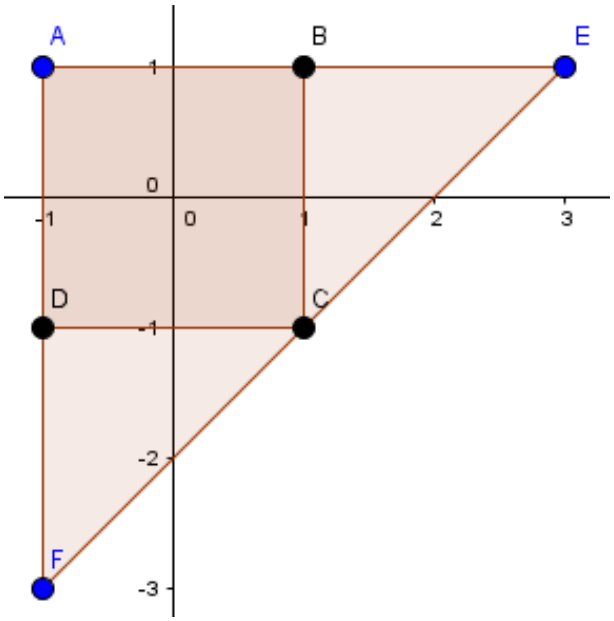
Väljastatava värvi arvutamine on analoogne punktvalgusallikaga.

3.6 Valguskaardi sujutamine stseeni (*lightmap blending*)

Valguskaardi sujutamise protsess on triviaalne: võetakse värvikaart ja valguskaart, korrutatakse need omavahel läbi ja väljastatakse algsele *render targetile*.

3.6.1 Lähtestamine

Kasutatakse staatilist *vertexpuhvrit* täisekraani suuruse ristküliku joonistamiseks. Puhvri andmed luuakse koheselt kärpekoordinaatruumis. Lisaks asukohale on vajalikud ka tekstuuri koordinaadid. Mikrooptimeerimise eesmärgil on puhvris nelja *vertexi* (ristküliku neli nurka) asemel kolm *vertexit*. Kuna kolm *vertexit* moodustavad kolmnurga, aga mitte nelinurga, on nende andmed deklareeritud järgnevalt: kui punktid A, B, C, D tähistavad ekraani nurkasid, on vaja luua kolmnurk A, E, F nii, et see kataks täielikult ekraani pinna. Ekraanist välja jääv ala kärbitakse graafika konveieril rasterdamise etapil.



Joonis 28. Täisekraani vertexpuhvri deklaratsioon

Joonistamisel kasutatakse topoloogiat *triangle stripe*, seega indekspuhver pole vajalik.

Analoogselt varjude maskimisele ja valgusallika joonistamisele, jäetakse sügavus-šabloontest lubamatuks.

Rasterdamisseisundil keelatakse *scissor test*. Samuti peab kärpimisviis olema rangelt vastupäeva, sest ei rakendata enam kasutaja poolt seadistatud teisendusmaatriksit.

Tabel 10. Valguskaardi stseeni sujutamise rasterdamisseisund

Nimetus	**Väärtus**
Kärpimisviis	Vastupäeva
Käärtest lubatud	Ei
Täiteviis	Tahke

Sujutamisseisund on järgnev:

Nimetus	**Väärtus**
Lubatud värvikanalid	Kõik

Värvi sujutamisfunktsioon	Liitmine
Värvi allika kordaja	Üks
Värvi sihtmärgi kordaja	Null

3.6.2 *Vertex shaderi* algoritm

Vertex shaderis vertexi asukohta ei teisendata, küll aga viiakse asukoha andmestruktuur standardsele kujule, kus w-komponendiks seadistatakse 1, et perspektiivjagamise käigus asukoht ei muutuks. Tekstuuri koordinaadid saadetakse edasi ilma muudatusteta.

```
vertexOutput.Position = float4(vertexInput.Position.x, vertexInput.Position.y, 0.0, 1.0);
vertexOutput.TextureCoordinate = vertexInput.TextureCoordinate;
```

3.6.3 Piksel *shaderi* algoritm

Piksel *shaderis* on vajalik värviproovi võtmine nii valguskaardist kui ka värvi kaardist vastavalt sisendi tekstuuri koordinaatidele. Võetud proovid korrutatakse ja väljastatakse.

```
float4 diffuseColor = DiffuseMap.Sample(TextureSampler, pixelInput.TextureCoordinate);
float4 lightColor = Lightmap.Sample(TextureSampler, pixelInput.TextureCoordinate);
return diffuseColor * lightColor;
```

3.7 Programmeerimisliidese dokumentatsioon

Teegi avalik liides koosneb kuuest klassist, millest neli hõlmavad valgusallika erinevaid tüüpe ja baasklassi. *PenumbraComponent* on teegi tuumaks ning haldab kogu valguskaardi loomisprotsessi ning selle sujutamist originaalstseeni. Põhiline viis, kuidas kasutaja komponendiga suhtleb, on seadistades valgusallikad (*Light*) ja kestad (*Hull*), mida rakendatakse valgusefektide simuleerimisel.

Klass: *PenumbraComponent* laiendab klassi *DrawableGameComponent*

Omadused:

Tabel 11. Klassi PenumbraComponent omadused

Nimetus	**Tüüp**	**Kirjeldus**

Debug	*Boolean*	Võta või väärtusta, kas silumine on lubatud. Silumise korral joonistatakse valgusallikate ja varjude geomeetria võrestikuna ning logitakse erinevaid süsteemi toiminguid silumisväljundisse.
AmbientColor	*Color*	Võta või väärtusta värv, mida kasutatakse välisvalguse simuleerimisel. Alfa kanalit ei arvestata.
Transform	*Matrix*	Võta või väärtusta 4x4 teisendusmaatriks, mida kasutatakse valgusallikate ja kestade andmestruktuuride teisendamisel stseeni koordinaatruumist kärpekoordinaatruumi.
SpriteBatchTransformEnabled	*Boolean*	Võta või väärtusta, kas komponent peaks automaatselt arvestama teisendusega, mida kasutab *MonoGame SpriteBatch* klass. Mugav kasutajale, kui kasutaja kasutab *SpriteBatch* klassi joonistamiseks.
Lights	*List<Light>*	Võta nimekiri simuleeritavatest kohtvalgusallikatest. Läbi selle nimekirja saab valgusallikaid lisada ja eemaldada.
Hulls	*List<Hull>*	Võta nimekiri simuleeritavatest kestadest. Läbi selle nimekirja saab kestasid lisada ja eemaldada.

Meetodid:

Tabel 12. Klassi PenumbraComponent meetodid

Signatuur	**Kirjeldus**

void Initialize()	Lähtestab komponendi ja laeb vajalikud ressursid kõvakettalt graafikakaardi mällu.
void BeginDraw()	Seadistab graafika konveierile *render targeti*, kuhu kasutaja soovib joonistada stseeni elemendid, mis peaksid olema valgustatud komponendi poolt. Kaadri vältel peab *BeginDraw* välja kutsuma enne *Draw* välja kutsumist.
void Draw(GameTime gameTime)	Teostab valguskaardi loomise ja selle sujutamise algsesse *render targetisse*. *BeginDraw* peab enne olema välja kutsutud.

Klass: *Light*

Omadused:

Tabel 13. Klassi Light omadused

Nimetus	**Tüüp**	**Kirjeldus**
Enabled	*Boolean*	Võta või väärtusta, kas valgusallikat arvestatakse valgusefektide simuleerimisel.
CastsShadows	*Boolean*	Võta või väärtusta, kas varjude maskimist teostatakse antud valgusallika puhul.
Position	*Vector2*	Võta või väärtusta valgusallika asukoht stseeni koordinaatruumis.
Origin	*Vector2*	Võta või väärtusta valgusallika kese joonistataval ristkülikul. Kese on normaliseeritud vahemikku 0 kuni 1, kuid võib ulatud ka vahemikust väljapoole. Väärtus (0, 0) tähistab ristküliku alumist vasakut nurka, väärtus (1, 1) ristküliku paremat ülemist nurka.

Rotation	*Single*	Võta või väärtusta valgusallika rotatsioon. Rotatsioon määrab ära, mis pidi valgusallikas on pööratud.
Scale	*Vector2*	Võta või väärtusta valgusallika suurus. Väärtuse x komponent defineerib suuruse piki x-telge, y piki y-telge. Valgusallikas, mis hõlmab 10 ühiku pikkuse ruudu kujulise ala vaadeldavas stseenis, peaks omama suurusväärtust (10, 10).
Color	*Color*	Võta või väärtusta valgusallika poolt kiiratava valguse värv. Alfa kanalit ei arvestata.
Intensity	*Single*	Võta või väärtusta valgusallika intensiivsus. Intensiivsus on faktor, millega lõppväärtus läbi korrutatakse, et leida lõplik valgusallika poolt väljastatav värv.
Radius	*Single*	Võta või väärtusta valgusallika keskme raadius. Raadius defineerib ära ala, mis kiirgab valgust. Väiksema raadiuse korral tekivad väiksemad poolvarju osad (vari on teravam), suurema raadiuse korral suuremad.
ShadowType	*ShadowType*	Võta või väärtusta kestade valgustamise tüüp. *ShadowType* on loetelu, kus väärtus *Solid* määrab ära, et kestasid ei valgustata valgusallika poolt, väärtus *Illuminated*, et kestasid valgustatakse.

Klass: *PointLight* laiendab klassi *Light*

Ei defineeri lisaks ühtegi omadust või meetodit.

Klass: *Spotlight* laiendab klassi *Light*

Omadused:

Tabel 14. Klassi Spotlight omadused

Nimetus	Tüüp	Kirjeldus
ConeDecay	*Single*	Võta või väärtusta prožektorvalguse koonuse hajuvusfaktor. Mida kõrgem väärtus, seda suurem on valgustugevus koonuse äärtes.

Klass: *TexturedLight* laiendab klassi *Light*

Omadused:

Tabel 15. Klassi TexturedLight omadused

Nimetus	Tüüp	Kirjeldus
Texture	*Texture2D*	Võta või väärtusta tekstuur, mida kasutatakse valgustugevuse määramiseks.

Klass: *Hull*

Omadused:

Tabel 16. Klassi Hull omadused

Nimetus	Tüüp	Kirjeldus
Points	*List<Vector2>*	Võta nimekiri kesta nurkadest. Läbi selle nimekirja on võimalik kesta nurki lisada ja eemaldada.
Enabled	*Boolean*	Võta või väärtusta, kas kesta arvestatakse valguskaardi loomisel.
Valid	*Boolean*	Võta, kas kest on kehtiv ehk kas kasutaja on defineerinud kesta nurgad õigesti. Kest on kehtiv, kui kestal on vähemalt kolm nurka ning kesta nurgad moodustavad lihtsa hulknurga (hulknurga, mille ükski külg ei ristu).

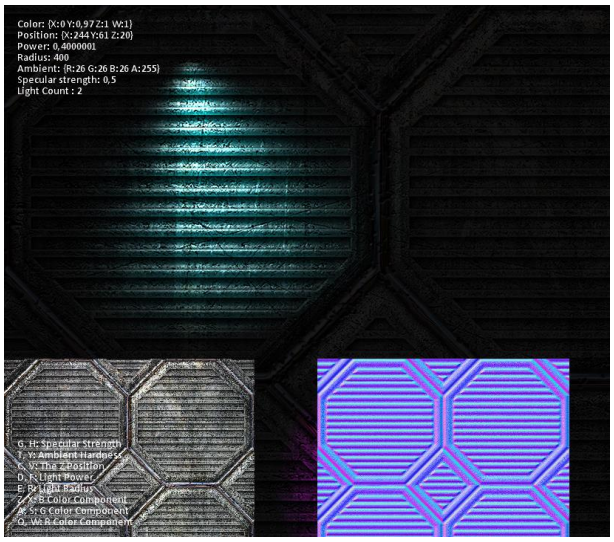
Position	*Vector2*	Võta või väärtusta kesta asukohta stseeni koordinaatruumis.
Origin	*Vector2*	Võta või väärtusta kesta null punkt lokaalses koordinaatruumis.
Rotation	*Single*	Võta või väärtusta kesta rotatsioon. Rotatsioon määrab ära , mis pidi kest on pööratud.
Scale	*Vector2*	Võta või väärtusta kesta suurus. Väärtuse x komponent määrab ära suuruskordaja piki x-telge, y komponent piki y-telge.

4. Arendusvaade

Käesolev osa räägib võimalikest edasiarendustest, mida teegi autor on plaaninud, kuid mis on jäänud realiseerimata nende keerukuse või muude ebasobivuste tõttu.

4.1 Normaalkaardistatud valgustus (*normal mapped lighting*)

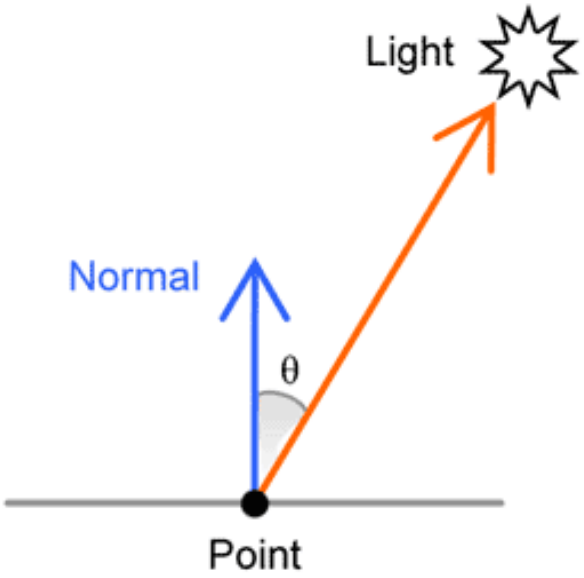
Normaalkaardistatud valgustus tähendab võimalust simuleerida valgusefekte selliselt, et kasutajale jäetakse 2-mõõtmelisest stseenist 3-mõõtmelise mulje.



Joonis 29. Normaalkaardistatud valgustus [14]

Taolise efekti rakendamine leiaks aset valgusallika joonistamisel (vt peatükk 3.5). Esmalt tuleks süsteemi muuta nii, et valgusallikad võimaldaksid kasutajal määrata ka kõrgust. Olgu selleks siis uus kõrguse omadus või muuta olemasoleva asukoha andmestruktuur *Vector2*'st *Vector3*'e, kus z-komponent tähistaks kõrgust. Suurimaks barjääriks oleks nõue, et kasutaja peab mingil moel võimaldama andmeid stseeni pinnanormaalide kohta.

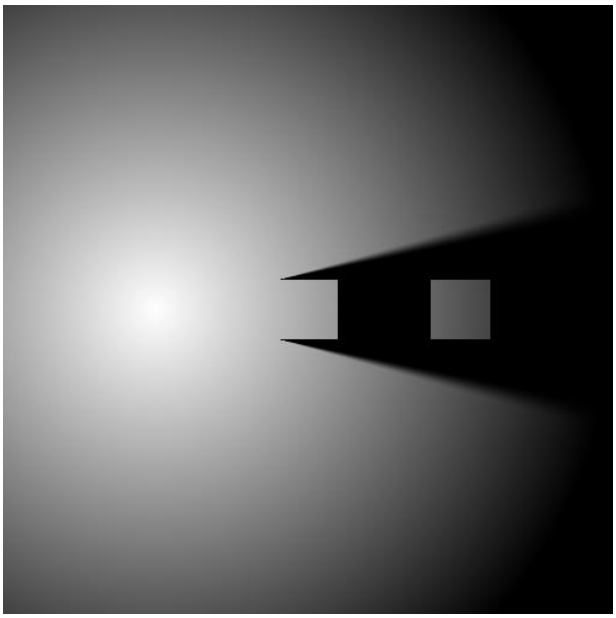
Valgusarvutus mingis punktis oleks triviaalne operatsioon. Kui on teada normaalvektor pinnapunktist valgusallikasse ja pinnapunkti enda normaalvektor, siis nende vektorite skalaarkorrutisest tulenevalt leitakse, kui palju valgust punktile rakendada.



Joonis 30. Valguse arvutamine pinnanormaali järgi [15]

4.2 Varjatud kestad (*occluded hulls*)

Kestade valgustamisel defineeriti valgustamiseks kaks võimalust: kas valgusallikas valgustab kestasid või mitte. Juhul kui valgustatakse, tekib nähtus, mis võib kasutajale tunduda soovimatu. See nähtus seisneb selles, et kuigi kest blokeerib valguskiirte levikut edasisele pinnale, siis ei bloki ta valguskiirte levikut kestadele, mis jäävad eelneva kesta taha.

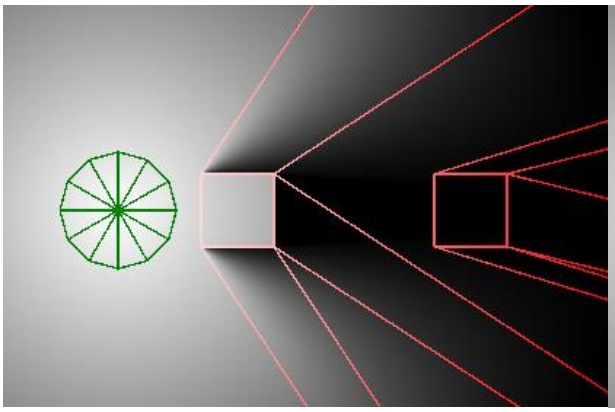


Joonis 31. Valgustatud kestade varjamatus

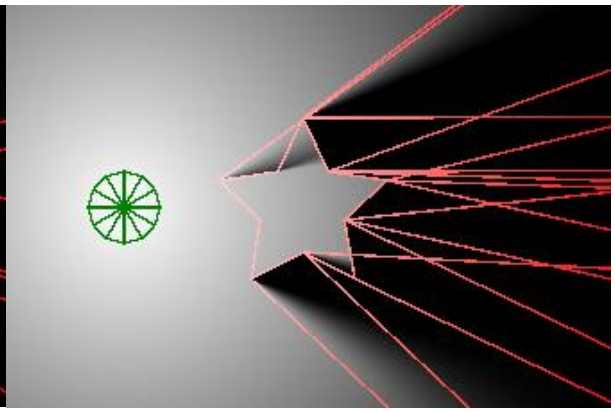
Eelnevalt jooniselt on näha probleem juhul, kui kasutaja soovib, et antud stsenaariumis parempoolne kest ei oleks valgustatud.

Lahendus probleemile oleks võtta kasutusele graafika konveieri poolt pakutav šabloontest (*stencil test*). Šabloontest teostatakse piksli puhul peale piksel *shaderi* etappi. Rasterdamisseisundis defineeritud šabloonväärtust võrreldakse šabloonpuhvris oleva väärtusega: kui võrdlus läbib, väljastatakse piksel; vastasel juhul kärbitakse. Idee seisneks selles, varjude maskimisel (vt peatükk 3.4) märkida varjude alla jäävad alad šabloonpuhvris ning kestade joonistamisel märgitud pikslitele mitte joonistada. Valgusallika joonistamisel (vt peatükk 3.5) nullitakse šabloonpuhvri andmed.

Eeltoodud lahendus paraku ei ole universaalne ehk lahenduse rakendamisel tekib uus probleem. Seoses viisiga, kuidas varju geomeetriat luuakse, saavad takistuseks nõgusad kestad (nõgus kest on kest, mille vähemalt üks nurkadest moodustab suurema kui 180 kraadise nurga). Probleemi illustreerib järgnev pilt, kus vasakul on näha korrektne käitumine kumera kesta puhul ning paremal ebakorrektne nõgusa kesta puhul.



Joonis 32. Varjatud kesta soovitud tulemus

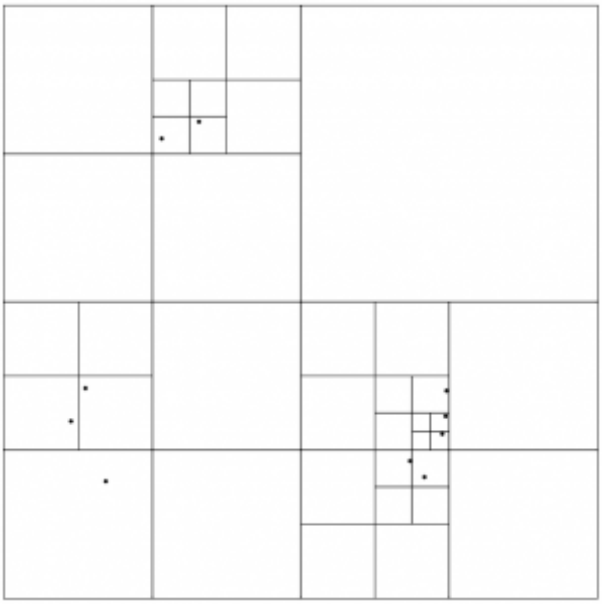


Joonis 33. Varjatud kesta iseärasusest ilmnev probleem

4.3 Ruumiline eraldamine (*spatial partitioning*)

Kui rakenduses on tegu näiteks väga suure mängumaailmaga, võib jõudlus käesoleva teegi puhul probleemiks osutuda, kui terve maailma valgusallikaid ja kestasid hoitakse korraga aktiivsena komponendis. Üldlevinud viis, kuidas sellise jõudlusprobleemi vastu võidelda, on struktureerida stseen väiksemateks osadeks ja uuendada/joonistada ainult alamosasid, mis jäävad ekraani vaatesse.

Üheks võimaluseks stseeni struktureerimiseks oleks kasutada nelinurkpuud (*quadtree*). Nelinurkpuu on lehest/lehtedest koosnev struktuur, mille puhul on paika pandud lävend, mitu elementi on maksimaalselt ühte lehte lubatud. Kui elementide arv ületab selle lävendi, tükeldatakse leht neljaks alamosaks. See tegevus toimub rekursiivselt kuni üheski lehes ei ole enam rohkem objekte, kui lävend lubab või kui maksimaalne puu sügavus on saavutatud.



Joonis 34. Nelinurkpuu [16]

Paraku taolise struktureerimise teegi poolne realisatsioon on raskendatud, sest ei ole teada täpse rakenduse tunnusjooned ega ka käsitletava maailma mõõtmed. Lisaks, kui stseen on selliste mõõtmetega, et ruumiline eraldamine hakkab ennast ära tasuma, on üldjuhul soov seda kasutada lisaks valgusefektide teegi elementidele ka teiste stseeni elementidega. Seetõttu on ruumilise eraldamise realisatsioon antud tööst välja jäetud mõttega, et selle kasutus- ja teostusdetailid peaksid olema paika pandud rakenduse autori (valgusefektide teegi kasutaja) poolt rakenduse tasemel.

5. Võimalikud rakendused

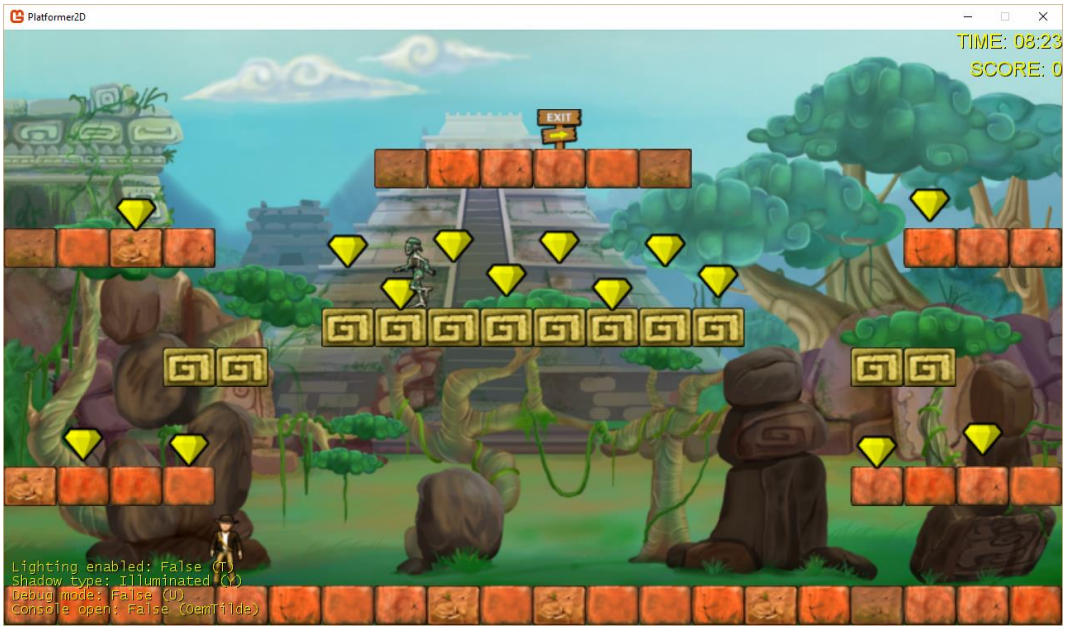
Lõppude lõpuks on teegi kasutamisvõimalused igaühe oma fantaasia vili. Autori arvates on parimad kandidaadid 2-mõõtmelised ülevalt alla või külgperspektiivis mängud. Sobivad kõiksugu žanrid ja stiilid. Probleemid tekivad, kui maailma kujutamisse hakatakse sisse tooma 3-mõõtmelisuse efekte, sest teegi käitumine 3-nda mõõtme puhul ei ole defineeritud. Seega ei ole soovitatav teeki kasutada näiteks isomeetrilise perspektiiviga mängude puhul.

Teegi rakendamise lihtsustamiseks on teek kättesaadavaks tehtud üle *NuGet* paketikanali.

NuGet on paketihaldur *Microsofi'*i arendusplatvormile kaasa arvatud *.NET* platvormile. *NuGet*'i klient võimaldab jagada ja tarbida pakette ning *Nuget Gallery* on keskne paketihoidla kõikidele pakettide tarbijatele ja autoritele ühiskasutamiseks. [17]

5.1 Rakendamine *MonoGame*'i näidismängul *Platformer2D*

Platformer2D on 2-mõõtmeline külgvaates platvormmäng, mille lähtekood on saadaval *MonoGame* avalikus *GitHub*'i koodihoidlas. Mängu eesmärk on demonstreerida arendajatele kuidas realiseerida *MonoGame* raamistikul karakteri liikumist, animeerimist, tuvastada kokkupõrkeid teiste objektidega, rakendada muusikat ja heliefekte, luua failipõhine tasemete süsteem, jms.



Joonis 35. Näidismäng

Teegi kaasamiseks projekti installeeritakse see läbi *NuGet* paketihalduskonsooli käivitades järgnev käsklus:

```
Install-Package Penumbra.MonoGame.WindowsDX -Pre
```

See laeb alla valgusefektide teegi (vajadusel ka *MonoGame* raamistiku teegi) ja lisab viited projekti faili.

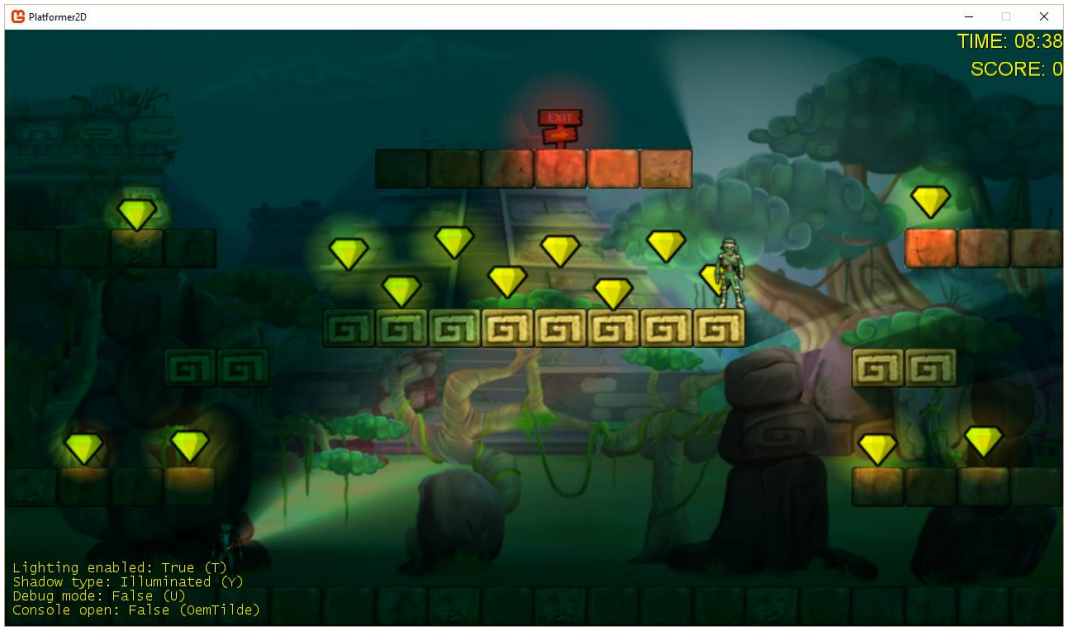
Rakenduse *Game* klassi konstruktoris luuakse uus valgusefektide teegi komponent ja hoiustatakse see muutujas hilisemaks kasutamiseks:

```
penumbra = new PenumbraComponent(this);
Components.Add(penumbra);
```

Enne stseeni elementide joonistamist, millele soovitakse valgusefekte rakendada, on oluline välja kutsuda meetod *BeginDraw*, mis seadistab *render target*'i nii, et valgusefektide komponendil oleks pärast võimalik ligi pääseda kasutaja poolt joonistatud andmetele:

```
penumbra.BeginDraw();
```

Sellega on baasseadistus tehtud. Järgnevalt lisatakse soovitud mänguelementidele valgusallikad ja taseme maastikule kestad. Kestade loomine on lihtsustatud, sest antud näidismängu puhul on tegemist korrapäraselt jaotatud ruutudega. Mängijale antakse prožektorvalgusallikas, mida hiire abil on võimalik suunata.



Joonis 36. Näidismäng koos valgusefektidega

6. Kokkuvõte

Käesoleva bakalaureusetöö eesmärgiks on leida lahendus probleemile, kuidas reaalajas simuleerida lihtsamaid valgusefekte 2-mõõtmelisel tasandil, kusjuures põhirõhk on täis- ja poolvarjude loomisel. Lahendus on *MonoGame* raamistikule loodud üldkasutatava teegi kujul.

Järgnevalt on loetletud käesoleva lõputöö tulemused ehk teegi poolt pakutav funktsionaalsus:

- Välisvalguse simuleerimine;
- Kohtvalgusallikate simuleerimine punkt-, prožektor- ja tekstuuritud valguse kujul;
- Varjude simuleerimine varje heitvate kestade näol.

Lisaks tulemustele, pakub töö ka võimalikud edasiarendused:

- Normaalkaardistatud valgustus, kus joonistatava pinna normaale arvesse võttes võimaldataks visuaalselt jätta 2-mõõtmelisest tasandist 3-mõõtmelise mulje;
- Varju jäävate kestade mittevalgustamine valgustatud kestade korral;
- Ruumiline eraldamine ehk valgusallikate ja kestade grupeerimine ja grupiviisiline haldamine jõudluse parendamiseks.

Summary

The goal of the Bachelor's thesis is to find a solution to a problem of how to simulate simple 2D lighting effects in real time. Main focus is on the generation of umbra and penumbra regions caused by blocking a light source. The result of this thesis is a general purpose library built on top of MonoGame framework.

The following is the resultant functionality provided by the library:

- Simulating ambient lighting;
- Simulating light sources as point, spot- or textured lights;
- Simulating soft shadows with both umbra and penumbra regions caused by the blocking of light by shadow hulls.

In addition to previous results, possible future improvements are also brought out:

- Normal mapped lighting, where considering surface normals would allow to visualize a 2D space as a 3D space;
- Not lighting shadow hulls occluded by other hulls;
- Spatial partitioning of lights and hulls in order to improve performance for high scale worlds.

Kasutatud kirjandus

[1] A. Russell, „Dark,“ [Võrgumaterjal]. Available: http://andrewrussell.net/dark/.

[2] J. Campbell, „Dynamic 2D Soft Shadows,“ 24 4 2013. [Võrgumaterjal]. Available: http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/dynamic-2d-soft-shadows-r3065.

[3] C. Harris, „Krypton XNA,“ 11 6 2013. [Võrgumaterjal]. Available: https://krypton.codeplex.com/.

[4] G. Stark, "Light," 12 5 2015. [Online]. Available: http://www.britannica.com/science/light#toc258391.

[5] „Lightcookies - Variation 2,“ [Võrgumaterjal]. Available: https://www.filterforge.com/filters/12167-v2.html.

[6] S. Halliday, „2D Dynamic Shadow Casting,“ 21 7 2010. [Võrgumaterjal]. Available: http://experimentalized.blogspot.com.ee/2010/07/2d-dynamic-shadow-casting.html.

[7] „Umbra, penumbra and antumbra,“ 14 10 2015. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Umbra,_penumbra_and_antumbra.

[8] „System Requirements | MonoGame,“ [Võrgumaterjal]. Available: http://www.monogame.net/documentation/?page=System_Requirements.

[9] „About | MonoGame,“ [Võrgumaterjal]. Available: http://www.monogame.net/about/.

[10] P. Roy, „DirectX/XNA Phase Out Continues,“ 30 1 2013. [Võrgumaterjal]. Available: https://ventspace.wordpress.com/2013/01/30/directxxna-phase-out-continues/.

[11] „GPU Gems - Chapter 34. GPU Flow-Control Idioms,“ Aprill 2005. [Võrgumaterjal]. Available: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter34.html.

[12] S. Lembcke, 2015.

[13] „DirectX 8 Programming Tutorial,“ [Võrgumaterjal]. Available: http://www.e-reading.club/bookreader.php/143437/Pike_-_DirectX_8_Programming_Tutorial.html.

[14] V. v. Soolen, „2D Spotlight implementation (HLSL + C#),“ 11 4 2012. [Võrgumaterjal]. Available: http://www.soolstyle.com/2012/04/11/2d-spotlight-implementation-hlsl-c/.

[15] J. Chapman, „Per-Pixel Lighting,“ 12 2 2010. [Võrgumaterjal]. Available: http://www.john-chapman.net/content.php?id=3.

[16] A. Beutel, „Interactive Quadtrees and Well-Separated Pairs Decomposition,“ 16 1 2012. [Võrgumaterjal]. Available: http://blog.alexbeutel.com/445/interactive-quadtrees-and-well-separated-pairs-decomposition/.

[17] „Nuget Gallery | Home,“ [Võrgumaterjal]. Available: https://www.nuget.org/.

[18] „e-Teatmik: IT ja sidetehnika seletav sõnaraamat,“ [Võrgumaterjal]. Available: http://vallaste.ee/.

[19] J. Varus, "discosultan/penumbra," [Online]. Available: https://github.com/discosultan/penumbra.