T A R T U Ü L I K O O L
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Informaatika eriala

Raimond-Hendrik Tunnel

Protseduuriline puude genereerimine

Bakalaureusetöö

Juhendaja: Konstantin Tretjakov, MSc
Kaas-juhendaja: Sven Laur, D.Sc. (Tech)

Autor: ..... “.....“ mai 2012
Juhendaja: ..... “.....“ mai 2012
Kaas-juhendaja: ..... “.....“ mai 2012

Lubada kaitsmisele
Õppetooli juhataja: ..... “.....“ 2012

TARTU 2012

Sisukord

1. Sissejuhatus

Arvutigraafikas on väga levinud puukujuliste visuaalide kujutamine. Sellised vorme läheb tihti vaja suures koguses ja seetõttu nõuab nende käsitsi modelleerimine palju ressurssi kunstnikelt. Puukujulised vormid võivad olla näiteks mõrad või praod seintes, jõestik maapinnal, muud sorti visuaalsed efektid ning loomulikult ka puud ise. Viimaseid on omakorda väga mitut liiki ning paljukesi koos moodustavad metsa.

Olenevalt virtuaalse keskkonna suurusest võib seal vaja olla kümneid, sadu või isegi tuhandeid puid. Metsa võib samuti kujutada kümnekonna puuga pannes need korduma ja luues niimoodi efekti paljudest erinevatest puudest. Rohkem unikaalsema ja varieeruvama keskkonna loomiseks peaks aga erinevaid puid olema rohkem – tõepoolest tuhandeid. On ebarealistlik lasta kunstnikel modelleerida tuhatkond erinevat puud. Siinkohal tulevad appi arvuti võime teha kiiresti matemaatilisi tehteid ning puude, või puukujuliste objektide, teatud matemaatiline korrapära.

Arvuti poolt algoritmi (kindlate reeglite) põhjal visuaalide ehk graafika loomist nimetatakse *protseduuriliseks genereerimiseks*. Algoritme, mis võimaldavad luua erinevaid puid, on erineva raskusastmega ja mitmeid. Käesolevas töös vaadeldakse ja realiseeritakse ühte traditsioonilisemat nendest. Selleks on formaalne grammatika nimega Lindenmayeri süsteem. L-süsteemi omadus genereerida iseendaga sarnaseid sõnesid on leidnud kasutust erinevate taimemudelite loomisel, kus grammatika poolt tuletatud sõna tähed vastavad graafiliste elementide joonistamise protseduuridele.

Siiski on viimase kümne aasta jooksul ilmunud mitmeid artikleid ja algoritme, kuidas muude meetoditega puid genereerida. Mõned nendest on mõeldud kiireks määramiseks, millist puud soovitakse, mõned aga orienteeritud kaamera kaugusest lähtuva detailsuse varieerimiseks. Käesolevas töös vaatleme peale L-süsteemi veel 1999nda aasta Paul Kruszewski artiklis "An algorithm for sculpting trees" esitletud meetodit puude genereerimiseks.

Töö jaotub kolmeks suuremaks osaks. Esimene neist käsitleb kasutatud tehnoloogiaid ja kirjeldab, kuidas neid on kasutatud puu genereerimise eesmärgi saavutamiseks. Välja on toodud nii põhjused vastavate tehnoloogiate eelistamiseks kui ka detailsemad kirjeldused näiteks programmisiseste funktsioonide väljakutsete jaoks. Selline lähenemine tutvustab

ülesande erinevaid komponente ja nende vahelisi seoseid. Teine osa sisaldab endas L-süsteemidel põhinevat algoritmi, selle realisatsiooni ja optimeeringut. Kolmas osa kirjeldab ja võrdleb põgusalt P. Kruszewski artiklis toodud algoritmi.

Osad on ülesehitatud tutvustamaks esmalt põhimõisteid ja seoseid, mis aitavad materjalist aru saada. Teine osa on detailsem realisatsiooni kirjeldus koos näitlikustava materjaliga soovi korral realisatsiooni korrata. Lõpus on tähelepanekud ja hinnangud, mis töö osas ilmnesid. Töö on kirjutatud pigem kirjeldavas vormis, et anda lugejale ülevaade ühest võimalusest, kuidas kolmemõõtmelise keskkonnaga seotud ülesannet (antud juhul puude genereerimist) saab lahendada. Sellise ülesehituse eesmärk on pakkuda ennekõike lugejale informatsiooni mõnede tekkitavate probleemide kohta, vaatenurka puude genereerimise ülesande lahendamiseks ning kohati ka inspireerida käesolevatest lahendustest edasi mõtlemist.

Lisas on välja toodud SVN repositooriumi aadress, mille kaudu on võimalik ligi pääseda töö jooksul kirjutatud tarkvara lähtekoodile.

2. Kasutatud tehnoloogia

2.1. C++ ja lahenduse taustkomponendid

Loodud raamistik kasutab programmeerimiskeelt C++. Selline valik sai tehtud, sest ühest küljest on tegu piisavalt madala keelega, et programm saaks olla kiire ning tööaega ei kuluks näiteks mõne kõrgema keele (Java, PHP) interpreteerimise peale. Huvi C++'i teegi Allegro vastu langetas otsuse just selle keele kasuks. Teisest küljest on tegu piisavalt kõrge keelega, et lubada mugavat objekt-orienteeritust ja sellest tulenevat lähtekoodi arusaadavat struktuuri.

Baasraamistikku (vt. Graafik 1) ehitades võeti aluseks OpenGL-i tutvustavas raamatus "OpenGL Programming Guide" toodud näite joonistada sfääre. Eesmärk oli saada sfääride joonistamine ja kolmemõõtmelisse virtuaalmaailma paigutamine piisavalt dünaamiliseks, et sfääride asemel võiksid olla hiljem puud.

Abiklassideks defineeriti vektorite ja värvide klassid (*Vector* ja *Color*). Samuti tuli luua valgusallikaid (*LightSource*), mille positsioon ja valguse värv oleksid lihtsasti muudetavad.

Oluline komponent oli kaamera, mille asukohta pidi olema võimalik muuta. Siinkohal tuli kasuks teek Allegro [1], mis võimaldas lihtsa vaevaga luua klahvivajutuste peale toimuvaid funktsioonide väljakutseid. Kaamera juures said realiseeritud järgnevad vabadused:

Liikumine (mööda telge)	**Kaamera telg**	**Klahv**
Edasi	Z	↑ või W
Tagasi	Z	↓ või S
Vasakule	X	A
Paremale	X	D
Pööramine (ümber telje)	**Kaamera telg**	**Klahv**
Vasakule	Y	← või Q
Paremale	Y	→ või E
Üles (ainult vaade)	X	Page Up
Alla (ainult vaade)	X	Page Down

Tabel 1: Põhilised rakenduse kaamera liigutamise sisendklahvid

See annab võimaluse liikuda mööda XZ-tasapinda (maapind) vabalt kõikjale ning muuta vaatenurka ükskõik millisesse suunda (vt Pilt 4). Selleks, et kaamerat tagurpidi ei saaks

pöörata, on piiratud ümber X-telje pööramine ära 90-kraadiga nii üles kui alla suunas. Too pööramine ei muuda kaamera sihte, vaid ainult vaatenurka, see tagab, et üles ja alla poole saab küll vaadata, kuid mitte liikuda.

2.2. Allegro

Mänguprogrammeerimise teek Allegro pakkus peale lihtsa klaviatuurisisendi kuulamise ka mugava võimaluse tekitada vajadustele vastava suurusega programmi aken. Allegro pakub vaikimisi võimalust kasutada pildi joonistamiseks kahte puhvrit[1], millest ühele joonistatakse uut kaadrit ning teist näidatakse vaatajale. Selle tõttu oli akna loomine, sinna joonistamine ja selle normaalselt vaatajale näitamine suhteliselt lihtne.

Allegroga toimus põhiliselt kasutajaliidese joonistamine ja kuvamine. Nimelt on programmi kasutajale vaja näidata erinevat infot hetkel käimasoleva töö kohta. Samuti ka konkreetseid seadete väärtusi (kas valgustus on sisse või välja lülitatud) ning näiteks kaamera asukohta ning vaatenurka maailmas (vt. Ekraanitõmmis 1 ja Ekraanitõmmis 2).

Teegil on olemas 3D graafikateegi OpenGL tugi, mis tähendab, et nende kahe integreerimine oli lihtsasti tehtav. Allegro poolt loodud kuvale tuli anda parameeter `ALLEGRO_OPENGL`, koodi tuli kaasata päisefail `allegro5/allegro_opengl.h` ning seejärel võis suvalisel hetkel kutsuda välja juba OpenGL-i spetsiifilisi käske 3D joonistamiseks.

Veel sai kasutatud Allegro taimereid, et kutsuda välja põhitsükli iteratsiooni 60 korda sekundi jooksul. Muul ajal kutsutakse funktsiooni `al_rest()`. See loob olukorra, kus rakenduse kaadrisagedus (*frames per second*) on maksimaalselt 60 ning muul ajal ei kurnata üleliigselt ressursse. Lihtsuse huvides on joonistamine ja muu rakenduse töö (näiteks liikumine) samas koodiplokis. Teine lahendus oleks olnud kasutada lõimi, et joonistamine, sisemised arvutused ja kasutaja interaktsioonid võiksid töötada üksteistest sõltumata.

Kasutajaliidese joonistamiseks on loodud *singleton*[2] programmeerimismustriga klass, mille üks ja ainus olem vastutab õigete joonistamisfunktsioonide väljakutsete eest. Analoogsed

1 Kahekordne puhverdus (*double buffering*) on tehnika ekraanile sujuvaks kaadrite joonistamiseks. Alates Allegro 5-st on puhvrite uuendamiseks käsk *al_flip_display()*, vaata ka: http://www.allegro.cc/manual/5/al_flip_display

2 *Singleton* programmeerimismuster – lähenemine kirjutada programmi privaatse konstruktoriga klass, mis hoiab endas klassimuutujana ühte ja ainsat olemit endast. Klass peab sisaldama ka klassifunktsiooni olemi kättesaamiseks. Lähenemine võimaldab defineerida objekti, mis on kättesaadav ja muudetav rakenduse globaalses skoobis, ja keelab ära rakenduses mitmete vastava klassi objektide loomise.

singleton klassid on ka rakenduse parameetrite (näiteks valgustust määrav tõeväärtus-muutuja) hoidmiseks ja veateadetega tegelemiseks.

2.3. OpenGL

Kolmemõõtmelise maailma programmeerimise teek OpenGL (*Open Graphics Library*) on üks levinumaid vastavaotstarbelisi teeke. Suurimaks konkurendiks OpenGL-ile on teek Direct3D, valik esimese kasuks tulenes selle vabavaralisest litsentsist ja multiplatvormsete võimaluste tõttu.

Õppimisel ja algoritmi realiseerimisel tugineti raamatule "OpenGL Programming Guide", hüüdnimega "The Red Book" [2]. Programmeerimine järgis suuresti olekumasina[3] mustrit, kus oleku defineerib viimasena kasutusele võetud maatriks. Näiteks igal 3D graafika esitlusel (*rendering*) tuleb esmalt laadida projektsioonimaatriks ja sellele määrata vaatenurga laius ning vaatevälja mõõtmed. Seejärel laadida `MODELVIEW` maatriks, millesse kirjutatakse kaamera asukoht, vaate vektor ning kaamerast üles suunav vektor (et üheselt defineerida kaamera positsioon ja orientatsioon ruumis). Kaamera ja vaate tekitamiseks kasutati lisateegi GLU (*OpenGL Utility Library*) funktsioone `gluPerspective()` ja `gluLookAt()`.

Graafikaobjektide joonistamiseks kasutati *glDisplayList* struktuuri, kuhu oli võimalik eelnevalt ära kompileerida objektide joonistamiseks vajalikud käsud. Seejärel esitluse korral tuli kutsuda ainult vastavat nimekirja indeksi järgi. See võimaldas eristada objektidel kahte põhilist protseduuri: ehitamine (funktsioon *Build()*) ja esitlus (funktsioon *Render()*). Ehitamisel arvutati objekti lokaalsed parameetrid ja lisati joonistamis-funktsioonid nimekirja. Ehitamist tuli kutsuda iga objekti jaoks programmi töö jooksul ühe korra objekti struktuuri loomise või muutumise ajal. Esitlust kutsuti aga igal iteratsioonil üle põhitsükli (iga kaadri korral).

Optimeerimiseks (vt ka peatükki Optimeering) pakub OpenGL veel peale nimekirjade kasutamise ka pügamist (*clipping*) ja praakimist (*culling*). Pügamise korral ei arvutata neid kujundeid, mis jäävad kaamera vaateväljast välja. Praakimine võimaldab defineerida baaskujundite (kolmnurgad) ühe külje, mida ei arvutata. Kui järgida tava, et vastupäeva järjekorras defineeritud kolmnurga tipud määravad kolmnurga pealmise külje (vt.

3 Olekumasina muster (*state-machine pattern*) – lähenemine kirjutada programmi objekt, millel kutsutakse välja rakenduse töö jooksul samu funktsioone, kuid funktsioonide tegevus sõltub objekti parameetrite väärtustusest. OpenGL rakendab kõiki väljakutsutavaid käske vastavalt viimasena laetud maatriksile ja paljudele üldisematele olekumuutujatele. [3]

Ekraanitõmmis 3) ja näiteks sfääri joonistamisel võtta arvesse, et pealmine külg jääks alati vaatama sfäärist väljapoole (vaataja poole), siis saab ilma nähtavate kadudeta välja praakida sfääri moodustavate kolmnurkade tagaküljed (sfääri sisse jääva pinna).

Valgusallikate poolt antav efekt käib koos joonistatavate objektide materjalide defineerimisega. Ühele valgusallikale saab defineerida hajusa (*diffuse*) ja terava (*specular*) valguse värvid ning tugevuse, tüüpiliselt on need ühel valgusallikal sama värvi või väga sarnased. Saab ka defineerida valgusallika poolt eraldatava ruumivalguse (*ambient*), mis liitub ükskõik kus oleva valgusega ja valgustab objekte kõikjalt.

Kui valgusallikas on määratud ja ta kiirgab valgust, siis nähtavaks muutumiseks peab see valgus pindadelt kaamerasse peegelduma. Selleks tuleb defineerida ka pindadele vastavad peegeldumisomadused. Nendeks on ruumivalguse peegeldumine (*ambient*), hajusa valguse peegeldumine (*diffuse*), terava valguse peegeldumine (*specular*), materjali enda helendumine (*emission*). Terava valguse peegeldumise juures saab määrata eraldi materjali läikivust (*shininess*). Tüüpiliselt tahame siin, et ruumivalguse ja hajusa valguse peegeldumised oleksid sama värvi (objekt on valguse käes ja varjus samat värvitooni, kuid lihtsalt erineva tumedusega). Terava valguse peegeldumine määratakse tavaliselt valgeks või halliks, et peegelduva valgusallika värvid jääksid samaks (lombilt peegelduv tänavalatern on ka peegelduses sama värvi). Materjali enda helendumist saab määrata siis, kui tahetakse, et lisaks valguse arvutamisele liidetakse ka otsa helendumise värv (näiteks punaselt helenduv liftinupp on erkpunane nii pimedas kui valges). Helendumist on rakenduses kasutatud valgusallikate asukohtadesse joonistatud markeerimisobjektide jaoks (vt. Ekraanitõmmis 2).

3. L-süsteemil põhinev algoritm

L-süsteem on formaalne grammatika, mille töötles välja bioloog Aristid Lindenmayer aastal 1968, et aksiomatiseerida bioloogilise arengu teooriat. Põhiline idee L-süsteemidel on kujutada keerulise struktuuriga objekte kasutades objektist lihtsamaid aksioome ja produktsioone (ümberkirjutamisreegleid), mis teisendavad lihtsama osa keerulisemaks. L-süsteemi põhiline erinevus Chomsky grammatikatega on produktsioonide rakendamine paralleelselt. Ühel iteratsioonisammul kirjutatakse produktsioonide järgi ümber kõik tähed, mille jaoks on produktsioon olemas. Selline lähenemine imiteerib bioloogilist kasvu, kus kõik objektid, mille edasiarenemine keerulisemaks on võimalik, muutuvad korraga.[4]

3.1. L-süsteem

Kõige lihtsam L-süsteemide klass on deterministlike ja kontekstivabade L-süsteemide klass.

Olgu

V - tähestik

V^* - kõigi tähestiku V põhjal moodustatud sõnade hulk

V^+ - kõigi tähestiku V põhjal moodustatud mittetühjade sõnade hulk

Kontekstivabaks L-süsteemiks ehk OL-süsteemiks nimetatakse kolmikut:

$G = \langle V, \omega, P \rangle$

kus

V - süsteemi tähestik

$\omega \in V^+$ - aksioom

$P \subset V \times V^*$ - lõplik produktsioonide hulk

Ühte produktsiooni $(a, \chi) \in P$ kirjutatakse $a \to \chi$ ja on eeldatud, et iga tähe $a \in V$ korral on vähemalt üks sõna $\chi \in V^*$ nii, et eksisteerib vastav $a \to \chi$. Kui ühtegi sellist produktsiooni ei ole hulgas *P*, siis eeldatakse, et seal eksisteerib samasusprodukt sioon $a \to a$.

OL-süsteem on deterministlik parajasti siis, kui iga $a \in V$ jaoks on täpselt üks $\chi \in V^*$ nii, et eksisteerib produktsioon $a \rightarrow \chi$. ([5], lk 2)

Üks deterministlik OL-süsteem defineerib üheselt genereeritava objekti struktuuri. Ilma tähestikku, reegleid või aksioomi muutmata saab varieerida ainult tehtavate iteratsioonide arvu ja interpretatsiooniga. See tähendab, et kui on defineeritud puud genereeriv deterministlik OL-süsteem, siis erinevaid puid saaks tekitada järgnevatel viisidel:

1. Vähendades või suurendades iteratsioone, mille käigus käesoleva sõna tähemärke asendatakse.
2. Lisades juhuslikkust OL-süsteemist saadud sõna interpreteerimisse ehk puu konstrueerimisse ja joonistamisesse.

Nende variantide korral loodud erinevad puud omaksid siiski sarnase struktuuriga alamosa. Esimesel variandil oleks suurema iteratsioonide arvuga genereeritud puul korduv alamosa väiksema iteratsiooni arvuga genereeritud puuga. Teisel variandil oleks puude üldine struktuur sarnane, kuid erineksid harude pikkused ja murdumisnurgad. Tulemuseks saadavad puud oleksid üksteisega liiga sarnased ja näiksid tehislikud. Selle lahendamiseks on olemas stohhastiline OL-süsteem, mis on järjestatud nelik:

$$G_\pi = \langle V, \omega, P, \pi \rangle$$

Tähestik *V*, aksioom ω ja produktsioonide hulk *P* on defineeritud samamoodi nagu tavalise OL-süsteemi korral.

Tõenäosusjaotus $\pi : P \rightarrow (0, 1]$ kujutab produktsioonide hulga produktsioonide tõenäosusteks. On eeldatud, et iga tähe $a \in V$ korral nende produktsioonide tõenäosuste, kus produktsiooni vasak pool on *a*, summa võrdub 1. ([5], lk 18)

See võimaldab luua olukorra, kus iteratsiooni käigus võivad ühest tähest moodustuda määratud tõenäosuste alusel erinevad sõnad.

Selleks, et L-süsteemi poolt genereeritud sõna põhjal oleks võimalik modelleerida puu (või mõni muu graafiline element), tuleb igale tähemärgile anda interpretatsioon. Tüüpilised interpretatsioonid kahemõõtmelisel juhul on toodud tabelis Tabel 2.

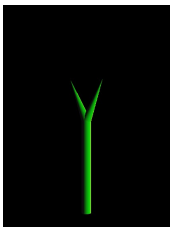
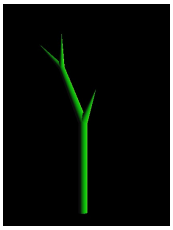
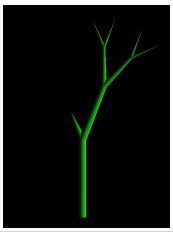
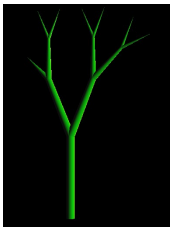
Täht	**Interpretatsioon**
F	Joonista puu segment praegusest asukohast otse ja liigu segmenti lõppu.
[	Salvesta praegune asukoht ja parameetrid pinusse
]	Võta uus asukoht ja parameetrid pinust
+	Pööra vasakule nurga δ võrra
-	Pööra paremale nurga δ võrra
δ – määratud pöördenurk, mis vastavalt realisatsioonile võib olla muutuv või fikseeritud	

Tabel 2: L-süsteemi tähemärkide iterpretatsioon kahemõõtmelisel juhul.

Produktsioonide hulk:

$$F \xrightarrow{0.5} F$$
$$F \xrightarrow{0.5} F\,[+F\,][-F\,]$$

Genereerib muu hulgas järgnevaid tulemusi kolme iteratsiooni korral:

F F F F[+F][-F]	
F F F[+F][-F] F[+F][-F][+F[+F][-F]][-F]	
F F[+F][-F] F[+F][-F][+F][-F[+F][-F]] F[+F][-F][+F][-F[+F][-F][+F[+F][-F]][-F[+F][-F]]]	
F F[+F][-F] F[+F[+F][-F]][-F[+F][-F]] F[+F[+F][-F[+F][-F]]][-F[+F][-F][+F[+F][-F]][-F[+F][-F]]]	

Tabel 3: Stohhastilise OL-süsteemiga genereeritud puud

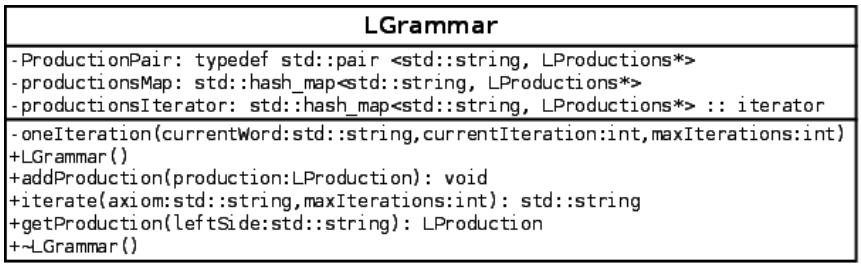
3.2. Algoritm

Eelmises peatükis toodud formaalse L-süsteemi definitsiooni järgi algoritmi realiseerimiseks on vajalik lahendada järgnevad ülesandeid:

- Grammatika ja produktsioonide hulga defineerimine.
- Nende põhjal etteantud iteratsioonide arvuga sõna genereerimine.
- Leitud sõnast puu struktuuri koostamine.
- Puu struktuuri põhjal puu joonistamine.

Nende alamülesannete täitmiseks koostati klassid, mis võimaldavad väljatoodud probleeme lahendada. Eesmärk oli eraldada semantiliselt erinevad tegevused alamosadeks, et saavutada parem ülevaade tehtavatest toimingutest ning pakkuda süstematiseeritud lähenemist vaadeldavasse probleemi.

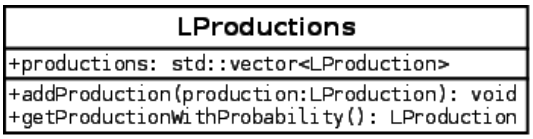
3.2.1. LGrammar



Klass, mis sisaldab endas produktsioone ja loogikat etteantud aksioomist ja iteratsioonide arvust tuletada sõna. Produktsioone hoitakse räsitabelis, mille räsifunktsioon on produktsiooni vasak pool. Räsitabeli kirjed on viidad klassi *LProductions* objektidele ja viimaste sees võib olla suvaline arv produktsioone. Funktsioon *getProduction(leftSide)* otsib räsitabelist soovitud vasaku poolega produktsioonide hulka ja küsib selle käest ühte tõenäolist produktsiooni. Juhul, kui räsitabelist ei leitud otsitava vasaku poolega kirjet, tagastatakse samasusproduktsioon.

Klassi eesmärk on hoida grammatikat ja sooritada etteantaval sõnal etteantud arv iteratsioone, kus iga tähemärgi asendus toimub vastavalt grammatikas defineeritud tõenäosuslikele produktsioonidele.

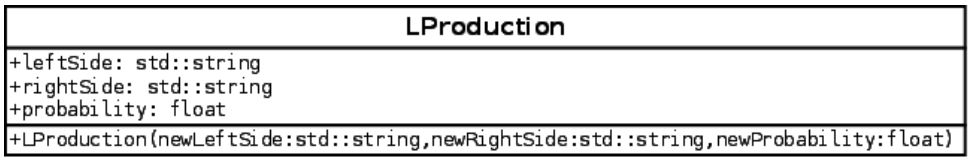
3.2.2. LProductions



Klass, mis hoiab endas mitmete produktsioonide kogumit. On eeldatud, et produktsioonid, mida selle klassi objekt hoiab, on ühesuguse vasaku poolega ja nende tõenäosuste summa on 1. Klass võimaldab sealt hulgast küsida tõenäosuse alusel leitavat produktsiooni. Tõenäosuse alusel produktsiooni leidmine toimub nii, et leitakse kõigepealt juhuslik arv vahemikust *[0, 1]* ja seejärel käiakse läbi järjest produktsioonid. Iga produktsiooni korral liidetakse produktsiooni tõenäosus juurde jooksvasse summasse. Kui jooksev summa on suurem juhuslikult leitud arvust, tagastatakse käesolev produktsioon. Selline lähenemine jaotab lõigu *[0, 1]* ära nii mitmeks osaks, kui on produktsioone, kusjuures iga osa on sama suur kui vastava produktsiooni tõenäosus, ja seejärel leitakse, millisele alamlõigule varem leitud juhuslik suurus sattus.

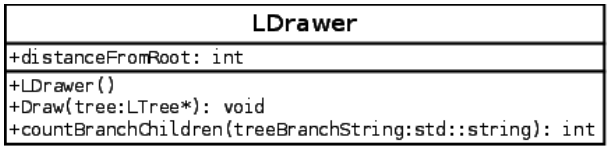
Keeles JavaScript on koostatud Kevin Roasti realisatsioon L-süsteemi interpretaatorist, mis kasutab mitte-stohhastilist varianti L-süsteemist ja produktsioone hoitakse massiivis, mille võtmed on produktsioonide vasakud pooled. [6] Käesoleva töö rakenduse *LProductions* klass pakub aga võimalust defineerida stohhastiline L-süsteemi produktsioone, mis võivad olla sama vasaku poolega.

3.2.3. LProduction



Klass, mis hoiab ühte produktsiooni koos tema vasaku ja parema poolega ning tõenäosusega.

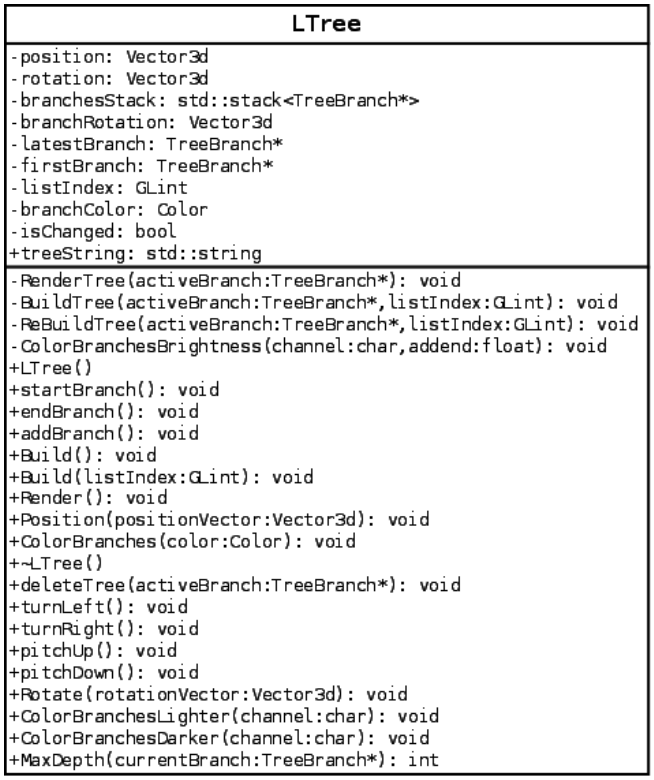
3.2.4. LDrawer



Klass, mis vastutab selle eest, et klassi *LTree* objektis määratud L-süsteemi poolt leitud

sõne sümbolite järgi kutsutakse välja *LTree* objekti õigeid funktsioone. Teisisõnu tegeleb see klass sõne põhjal objektide loomisega. Klassil on veel funktsioon, mis suudab sõne-töötluse järgi leida iga tipu alamtippude arvu.

3.2.5. LTree



Klass, mis seob omavahel L-süsteemi puu ja OpenGL-is joonistatava puu. Eesmärk on võimaldada puu struktuuri ehitamist ja struktuuri põhjal puu joonistamist. Eelmises alampeatükis kirjeldatud klass *LDrawer* saab sisse käesoleva klassi objekti viida ja kutsub välja sellest puu ehitamiseks vajalikke funktsioone. *LTree* klass vastutab, et iga puu segmendi puhul genereeritakse uus *TreeBranch* klassi objekt ja orienteeritakse see õigesse suunda.

Klassis olev pinu *branchesStack* vastutab selle eest, et pärast iga haru lõppu on võimalik naasta selle segmendi juurde, kust hargnemine alguse sai, et tollest kohast jätkata ülejäänud puu konstrueerimist. Pinu kasutavad funktsioonid *startBranch()* ja *endBranch()*, mida kutsutakse välja, kui L-süsteemi sõnes esinevad märgid "[" ja "]".

Funktsioonid *Render()* ja *Build()* kutsuvad omakorda välja funktsioone *RenderTree()* ja *BuildTree()*. Viimased vastutavad selle eest, et terve puu joonistamiseks vajalikud OpenGL-i käsud kompileeritakse õigesse nimistusse ja kutsutakse sealt välja. Kuna iga

segmendi joonistamiseks vajalikud käsud on klassi *TreeBranch* objektides, siis kompileerimisel käiakse puu läbi süvitsiotsinguga ja kutsutakse välja iga segmendi *Build()* funktsiooni.

Praeguses realisatsioonis hoitakse ühe puu joonistamiskäske ühes OpenGL nimistus. See tähendab, et puu kujutamiseks on vaja välja kutsuda ainult ühte nimistut ja algoritm on seetõttu kiirem. Teisest küljest on puu muutmiseks vaja uuesti kompileerida terve nimistu isegi siis, kui muudetakse ainult ühte segmenti.

3.3. Tulemus

L-süsteemide juures on soovitava puu defineerimine L-süsteemi produktsioonide alusel keeruline ülesanne. ([7], lk 2-3) Töö raames sai läbi proovitud teatud hulk produktsioone ja vaadeldud nende poolt tekitatud puid. Mõned huvipakkuvamad juhud on järgnevalt esitatud. Valik on produktsioonide arvu järgi kasvavas järjekorras ja iga näite juures on analüüsitud saadud tulemust. Lisaks varem mainitud kahemõõtmelise juhu sümbolitele (Tabel 2) on kasutusel ka sümbolid "^" ja "&", mis määravad pööramisi ümber X-telje.

Genereerides puu üheainsa produktsiooniga:

LProduction("F", "F[^F[+F][-F]][&F[+F][-F]]", 1);

ja iteratsioonide arvuga 3 aksioomist *F*, oli tulemus tasakaalus, kuid ebaloomulikult sümmeetriline ning puudus juhuslikkus (vt Ekraanitõmmis 4).

Proovides genereerida puud kahe järgneva produktsioonidega:

LProduction("F", "F[^F[+F][-F]]", 0.5);

LProduction("F", "F[&F[+F][-F]]", 0.5);

ja iteratsioonide arvuga 3 aksioomist *F*, oli tulemus juhuslik ja ebasümmeetriline, kuid tasakaalust liiga väljas. Puu kaldus sinna suunas, milline pööre (produktsioon) rakendati esimesena (vt Ekraanitõmmis 5).

Genereerides puu järgnevate produktsioonidega:

LProduction("F", "H[+G][-G]", 1);

LProduction("G", "H[&G]", 0.3);

LProduction("G", "H[^G]", 0.3);

LProduction("G", "^H[+G][-G]", 0.2);

LProduction("G", "&H[+G][-G]", 0.2);

LProduction("H", "H", 0.9);

LProduction("H", "H[&F]", 0.05);

LProduction("H", "H[^F]", 0.05);

ja iteratsioonide arvuga 7 aksioomist *F*, jäi tulemus teistega võrreldes parem. Puu ei olnud üldiselt ebaloomulikult sümmeetriline ega kaldunud ühte domineerivasse suunda (vt Ekraanitõmmis 6).

Oluline on see, et erinevalt eelnevatest produktsioonidest on kasutatud siin kolme sümbolit puu segmendi joonistamiseks (*F*, *G* ja *H*). See võimaldab lihtsasti ära määrata kolm erinevat segmenti, millel on erinevad võimalused edasiarenemiseks.

Produktsioon *LProduction("F", "H[+G][-G]", 1);* määrab ära selle, et esimesel sammul toimub kohe hargnemine. Luuakse segment *H*, mis hilisemate produktsioonide järgi selgub, et on vähe muutuv. Tekkiv hargnemine jätkab puud kahes alati fikseeritud suunas. Kuna selle produktsiooni tõenäosus on 1, siis alustades aksioomist *F* toimub alati puudel samasugune hargnemine. On küll väike tõenäosus, et puu esimeseks segmendiks saav *H* võib omakorda hiljem muunduda, kuid üldiselt tekitab see produktsioon siiski ebaloomulikku sümmeetriat ja korduvust erinevatel puudel.

Produktsioonid *LProduction("G", "H[&G]", 0.3);* ja *LProduction("G", "H[^G]", 0.3);* määravad, et segment *G* võib endast maha jätta vähe muutuva segmendi *H* ja tekitada ühe haru *G* üles või alla suunda.

Produktsioonid *LProduction("G", "^H[+G][-G]", 0.2);* ja *LProduction("G", "&H[+G][-G]", 0.2);* käituvad sarnaselt, ent koheselt toimub pöördumine üles või alla suunda, maha jääb vähe muutuv *H*, ning seejärel tekib kaks hargnemist *G*-ks paremale ja vasakule suunda.

Kolm viimast produktsiooni *LProduction("H", "H", 0.9);*, *LProduction("H", "H[&F]", 0.05);* ja *LProduction("H", "H[^F]", 0.05);* defineerivad segmendi *H* vähe tõenäolise muutumise. Sellise omadusega segment tasakaalustab puu liiga paljude ning liiga väheste

hargnemistega puude vahel. Juhul, kui iga segment suure tõenäosusega hargneb mitmeks, võib tekkida olukord, kus puu on liiga tihedalt koos ja võib ka juhtuda, et iga järgnev hargnemine on eelnevaga samasugune ning seetõttu kaotab puu enda loomulikkuse. Näiteks, kui eemaldada produktsioonid vasaku poolega *H* ja iga *H* teiste produktsioonide paremas pooles asendada *G*-ga, tekib liiga rohkete hargnemistega tihe puu (vt Ekraanitõmmis 7). Sellise olukorra leevendamiseks ongi käesolev produktsioon, mis tõenäosusega 0.9 on samasusproduktsioon ja vastasel korral tekitab üles või alla poole suunduva uue haru *F*.

3.4. Optimeering

Rakenduse eesmärk on säilitada kaadrisagedust töö ajal 50-60 vahel. Samas suudab arvuti vastavalt oma jõudlusele ära töödelda ainult limiteeritud arvu graafilisi objekte ja neid joonistada. Sellel põhjusel oli suuremate või rohkemate puude genereerimise ja joonistamise korral märgata kaadrisageduse langust (vt Tabel 4). Konkreetne piir sõltub arvuti spetsifikatsioonist, kuid teatud määrani optimeerimine on siiski levinud jõudlus-võimsusega arvutitel kaadrisageduse hoidmiseks kasulik ja vajalik.

Tüvi-koonuseid	Ilma optimee-ringuta	Praaki-misega	Kvaliteedi indeks 1	**Arvuti spetsifikatsioon** *CPU: Intel Core 2 Duo E6400, 2.13Ghz* *RAM: 3GB* *Video: ATI Radeon HD 5570, 1GB* *OpenGL versioon 4.1.10524*
2401 (1 puu)	**35**	**36**	**60**	

2058 (6 puud)	**41**	**42**	Juht	1						2						3					
			Puu	A	B	C	D	E	F	A	B	C	D	E	F	A	B	C	D	E	F
			Kavaliteedi indeks	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2
			Kaadrisagedus	**49**						**55**						**60**					

Tabel 4: Kaadrisagedused erinevate optimeeringutega

Teisest küljest seavad piiri ka kasutatud OpenGL käsunimistud, sest nendes on võimalik hoida ka piiratud koguses videomälu võrra käske. Seega veel efektiivsema tulemuse jaoks oleks vajalik kasutusele võtta käsunimistute asemel OpenGL-i tipumassiivid või VBO-d (*vertex buffer object*). Nimetatud lähenemised jäävad töö raamidest välja, sest põhiline eesmärk on puu genereerimine, mitte võimalikult optimaalne rohkearvulise koguse puude genereerimine. Samas mõõdukas optimeering osutus töö raames mõistlikuks. Samuti pakub optimeeringu käsitlemine juhiseid teistsuguse lähteülesandega, kuid kindla

kaadrisageduse hoidmise suhtes tundliku tarkvara jaoks.

3.4.1. Praakimine (*culling*)

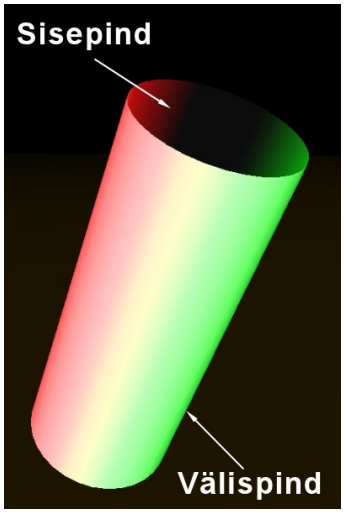
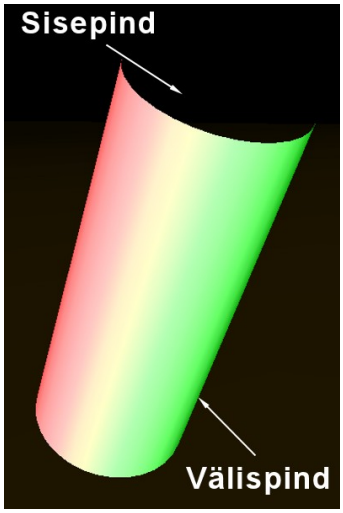
OpenGL pakub kiiret viisi määramaks, et arvutataks ja joonistataks ainult määratud kujundite välispindu. Kolmemõõtmelised kujundid hõlmavad enda alla pinnaga määratud ruumala ja suure tõenäosusega ei ole vajalik selle ruumala sisepinda vaatleja jaoks käsitleda. Isegi näiteks sein on kõige lihtsamas mõttes risttahukas, mille välispind (tapeet, värv) on vaatleja jaoks oluline, kuid seina sisse jääv pind mitte.

Käesolevas töös on enim kasutatud kujundiks tüvikoonus, mille erijuht on silinder. Kõik puude segmendid on määratud kõrguse ja kahe raadiusega tüvikoonused. Puu segmentide juures on oluline, et arvuti ressurss läheks välispinna arvutamisele ja kujutamisele, kuid sisepind jääb alati vaatleja jaoks nägematuks.

Praakimise saab OpenGL-is sisse lülitada järgnevate käskudega:

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
```

Esimene määrab, et pinnatahkude praakimine võetakse kasutusele ning teine määrab, et praagitakse välja pinna tagumised tahud. Kolmemõõtmelised kujundid on määratud nende pinnaga, millel on kaks poolt: eesmine ja tagumine. Pinna eesmine ja tagumine pool määravad ära kujundi sise- ja välispinna. Täpsemalt määrab selle ära pinna normaalvektor, mille siht on pinnaga risti ning suund alati pinna eesmisest poolest välja.

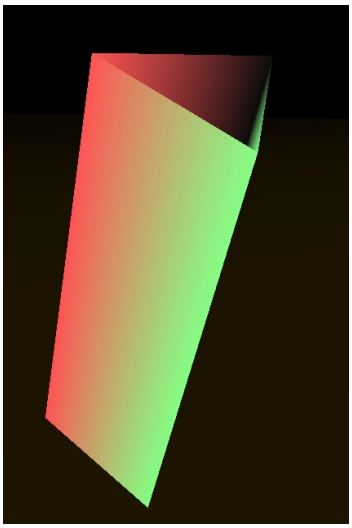
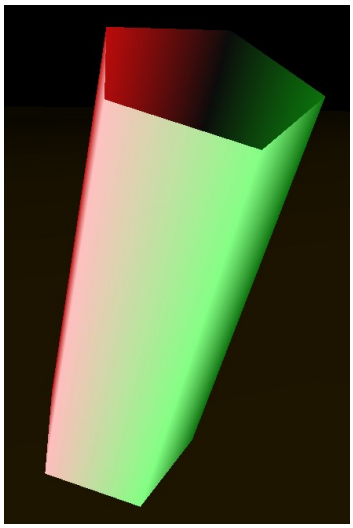
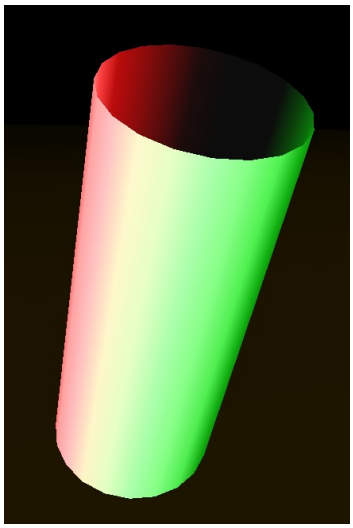
Ilma praakimiseta	Tagakülje praakimisega
	

Tabel 5: Praakimine

3.4.2. Viilud ja kihid

OpenGL-i käsk tüvikoonuse joonistamiseks soovib kaheks viimaseks parameetriks viilude (*slices*) ja kihtide (*stacks*) arvu. Need kaks parameetrit määravad ära, kui palju tehakse pinna alamjaotusi objekti Z-telje ümber ja Z-telge mööda. Tüvikoonuse puhul läheb Z-telg tüvikoonuse pikisuunaga paralleelselt. Selline lähenemine on tingitud sellest, et OpenGL-is ei ole tüvikoonuse primitiivi, vaid tüvikoonus pannakse kokku nelinurkadest, mille moodustavad komplanaarsed[4] tipunelikud (*quad*). Viilud ja kihid määravad ära kui detailne tüvikoonuse lähend genereeritakse (vaata Tabel 6).

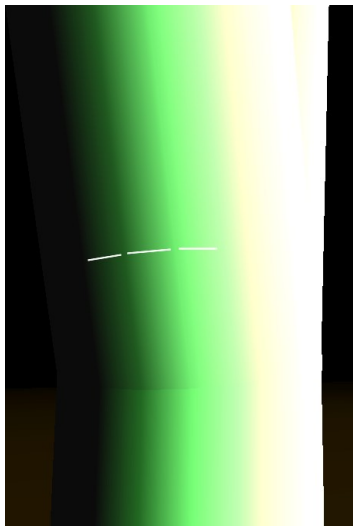
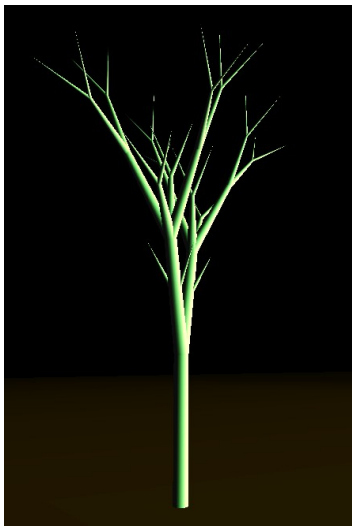
Selline lähenemine tüvikoonuse joonistamisesse annab võimaluse optimeerida vastavalt kaamera kaugusele joonsitatavast objektist (näiteks puust). Kaugemal olev objekt kujutub väiksemaks pikslitehulgaks kui sama suur ent lähemal olev objekt. See tähendab, et teatud kaugusel oleva tüvikoonuse korral ei ole ekraanil märgata vahet kümneks, kahekümneks või viiekümneks viiluks jaotatud tüvikoonustel. Samas iga viil defineerib ühe kihi korral ära ühe nelinurga, mille arvutamise jaoks kulub ressursse. Seega on otstarbekas lähendada lähemal asuvad kujundid täpsemalt ja kaugemal olevad kujundid vähem täpsemalt.

Kolm viilu	Viis viilu	Kakskümmend viilu
		

Tabel 6: Tüvikoonuse lähendamine viiludega

Järgneval näitel (Tabel 7) on genereeritud puu iga segmendi tüvikoonuse joonistamisel kasutatud kahtekümmet viilu. Lähedalt vaadates on näha üleminekud viilude vahel, kuid kaugemalt ei ole viilude olemasolust võimalik silma järgi aru saada.

4 Komplanaarsed – samal tasandil asuvad

Puu lähedalt vaadatuna	Puu kaugelt vaadatuna
	

Tabel 7: Puu kahekümne viiluga lähedalt ja kaugelt

Sellest omadusest lähtuvalt on töös genereeritud iga puu jaoks viis nimistut, kus puid moodustavatel tüvikoonustel on erinev arv viile ja kihte. Kui kaamera asub konkreetsest puust kaugel, kasutatakse nimistut, kus on vähem viile ja kihte. Lähemal asuvate puude korral kasutatakse rohkemate viilude ja kihtide arvuga nimistut. Konkreetsed viilude ja kihtide arvud ning kaugused sõltuvad üldjuhul rakenduse resolutsioonist, arvuti võimsusest ja kasutaja eelistustest. Antud töö raames on koostatava tüvikoonuse käsk:

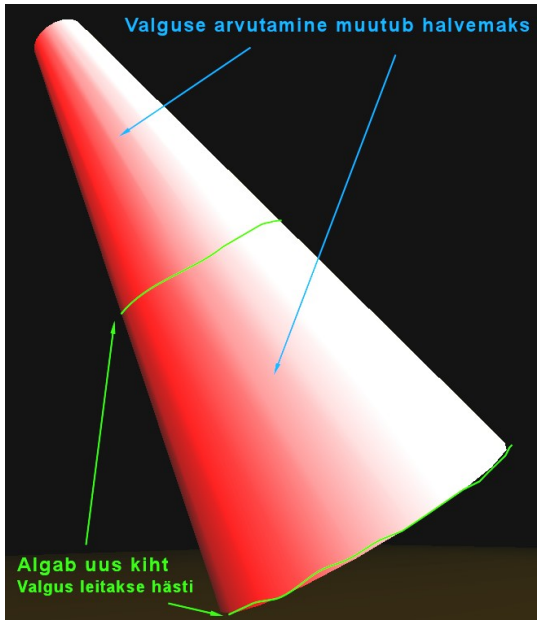
```
gluCylinder(
      quadObj, baseRadius, topRadius, height,
      (int)(32*quality),
      (int)(6*quality)
);
```

Muutuja quality saab väärtusi järgneva valemi järgi: `1.0/(sqrt(i+1.0)), i = 0 kuni 4`. Konkreetsed viilude ja kihtide väärtused on ära toodud järgnevas tabelis (vt Tabel 8).

`i`	`quality`	Viile (*slices*)	Kihte (*stacks*)
0	1	32	6
1	~0.707	22	4
2	~0.577	18	3
3	0.5	16	3
4	~0.45	14	2

Tabel 8: Erinevates nimistutes olevate tüvikoonuste viilude ja kihtide arvud

Eespool sai kirjeldatud viilude tähtsusest tüvikoonuse erijuhu – silindri lähendamisel. Kihtide arv määrab ära alajaotuste arvu mööda Z-telge. Liiga väikse kihtide arvu korral võib joonistatav objekt jääda mittesujuv isegi, kui viilude jaotus oleks muidu piisav.



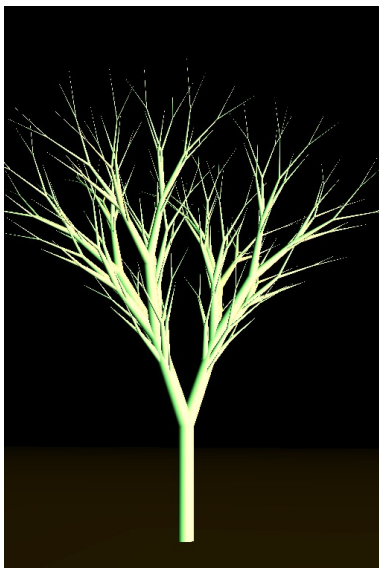
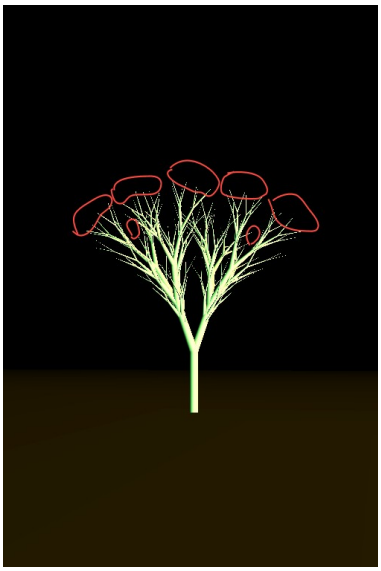
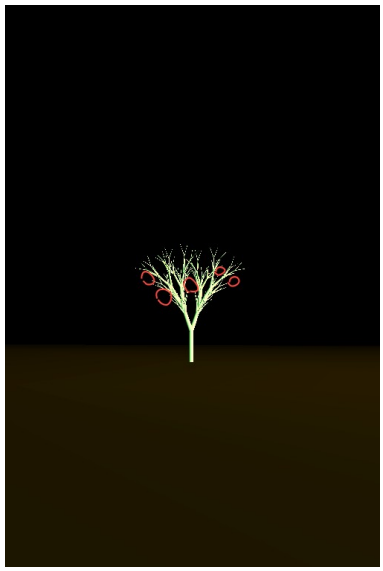
Pilt 1: Tüvikoonus kahe kihi ja kahekümne viiluga

Pildil Pilt 1 on näha, kuidas kahekümne viiluga ja erinevate raadiustega tüvikoonus muutub kahe kihi juures teatud aladelt ikkagi triibuliseks. Põhjus sellise efekti ilmnemiseks on selles, et tüvikoonusele määratakse normaalid tippudele. Antud objekt koosneb 40-st hulk-nurgast (20 viilu * 2 kihti). Nendel hulknurkadel on tipud koonuse alumises ääres, ülemises ääres ja keskel. Põhimõtteliselt võib mõelda, et kujund on kokku pandud kahest 20-viilulisest tüvikoonusest. Hulknurkade tippudele määratud normaalid võimaldavad nendes punktides valguse õiget arvutamist. Kuna ülemise ja alumise osa keskel ei ole tippe, muutub seal valguse arvutamine halvemaks, pind muutub triibuliseks. Saab arvata, et ehk on lihtsalt tüvikoonuse viile liiga vähe, kuid tegelikult viilude lisamine ei anna kõige optimaalsemat efekti. Määrates samale tüvikoonusele 200 viilu, jäi tulemus lähedalt vaadates endiselt triibuliseks. Samas jättes viilude arvu 20-ks, kuid suurendades kihtide arvu 6-ni, oli tulemus vägagi sujuv. Esimesel juhul loodi 200 hulknurka ja teisel juhul loodi 120 (20*6) hulknurka. Seega on piisavalt väikse pikkuse ja suurte raadiustega tüvikoonuse korral saavutada sujuvam ja optimaalsem tulemus kihtide arvu tõstmisega kui viilude arvu tõstmisega.

3.4.3. Segmendid

Viile ja kihte käsitlevas peatükis sai mainitud, et iga puu jaoks genereeritakse viis erinevat nimistut, kus igal segmentidel on erinev arv viile ja kihte. Põhimõtteliselt saaks mainitud lähenemist veel optimeerida selliselt, et määratav viilude ja kihtide arv võtaks arvesse puu segmendi (tüvikoonuse) raadiusi ja pikkust. Ehk siis, kui raadiused on väiksemad, genereeritakse vähem viile, kui pikkus on lühem, genereeritakse vähem kihte.

Kirjeldatud optimeering oleks edasiarendus praegusest, kus piisavalt väikeste raadiustega segmentide korral vastavaid segmente nimistusse ei panda. Tekib olukord, et kui puu on piisavalt kaugel, et tema kõige väiksemad oksad paistaksid imepisikesed, siis neid väiksemaid oksi üldse ei joonistatagi. On võimalik, et selline efekt lõhub virtuaalmaailma illusiooni, kui puude oksad ühel hetkel vaatleja liikudes tekivad ja kaovad. Selle vältimiseks oleks võimalik tekitada keskkonda atmosfäärinähtusi (näiteks udu), mis väikeste objektide tekkimist ja kadumist mingil määral varjaksid. Paljudes mängudes just nii tehtud ongi, et kauguses on udu ja sellest ilmuvad aeglaselt lähenevad objektid. Selle lahendusega koos saaks ehk objekte tekkima ja kaduma määrata läbipaistvusastmega, et need protsessid ei oleks vaatlejale niivõrd märgatavad. Optimeeringu seisukohast on aga hea, et kui objekt on nii väike, et ei oma üldises plaanis tähtsust, siis seda ei joonistata.

Puu lähedalt	Puu keskmiselt kauguselt	Puu kaugelt
		

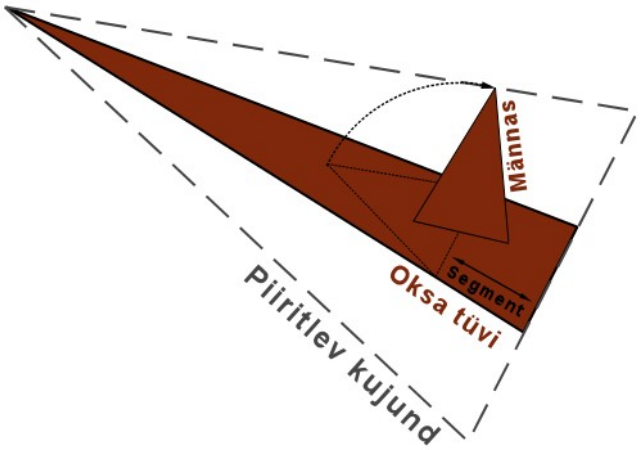
Tabel 9: Puu väiksemate segmentide mitte joonistamine vastavalt kaugusele. Punasega on ära märgitud mõningad mitte joonistatud segmendid.

4. Puu vormimise algoritm

4.1. Ülevaade

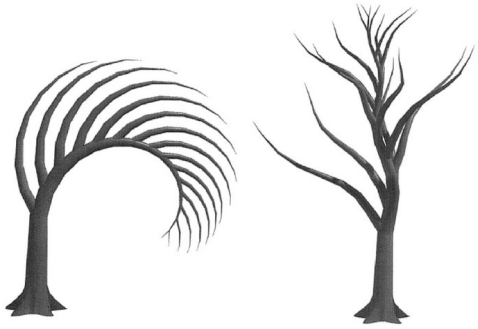
Paul Kruszewski artikkel "An Algorithm for Sculpting Trees" ilmus 1999ndal aastal ajakirjas Computers & Graphics. Artiklis kirjeldatakse algoritmi, mille lähenemine puude genereerimisele ei ole mitte puu kasvatamine juurest, nagu seda teevad enamikud muud algoritmid, vaid pigem vormimine väljastpoolt. Artikli sissejuhatuses on võrreldud Kruszewski algoritmi paari teise algoritmiga, seal hulgas ka Lindenmayeri süsteemidel põhineva lahendusega.

Puu vormimise algoritm üldjoontes käsitleb igat oksa kahe osana: oksa tüvi ja piiritlev kujund, mille sisse peavad mahtuma tüvest lähtuvad alam-oksad. ([7], lk 3) Sellest tulenevalt tegeleb algoritm puu vormimisega ehk suurema kujundi sisse väiksema loomisega. Iga oksa korral, mille tase on väiksem kui puu maksimaalne tasemete arv ning pikkus pikem kui minimaalne oksa pikkus, genereeritakse alam-oksad pikki oksa pikkust. Alam-oksad on grupeeritud männasteks (*whorl*), mis eralduvad ülem-oksast ja peavad ära mahtuma ülem-oksa piirkujundi sisse (vt Pilt 2).



Pilt 2: Oksa tüve jaotamine segmendiks ja sealt männase eraldumine piiritleva kujundi sisse.

Männased genereeritakse oksa mööda tipu poole kahaneva vahega. Selle vahe järgi jaotatakse ka ülem-oks omakorda segmentideks. Pööramised toimuvad männase murdumiseks ülem-oksast eemale ja segmentidel ümber nende peatelje. Oksa tüvikoonuse põhjaraadiuste arvutamiseks määratakse ära tüvikoonuse kahanemisnurk igal tasemel ning selle järgi toimub raadiuste arvutamine. ([7], lk 4-5)



Pilt 3: Segmentide pööramine.
Vasakul – kõiki segmente pööratakse ühes suunas. Saadakse ühele poole kaldu tüvi.
Paremal – kõiki segmente pööratakse ümber kesktelgede. Saadakse spiraaljas tüvi.
Pilt on pärit Kruszewski artiklist "An algorithm for sculpting trees". [7]

Kuna oksad ei lähe puudel sirgu, siis iga oksa segmendi korral pööratakse segmenti tema põhiteljest välja määratud nurga võrra. Sellise pööramise tulemusena jäävad puu oksad ühele poole (vastavalt määratud nurgale) kaldu. Lahenduseks tehakse veel teine pööramine ümber segmendi põhitelje (vt Pilt 3). ([7], lk 5) Selline lähenemine genereerib suhteliselt loomuliku puu põhistruktuuri (spiraali) kasutamata sealjuures juhuslikkust.

Artikkel pakub ka lineaarset lähenemist okste tüvikoonuste tahkude arvu vähendamiseks vastavalt oksa suurusele ja vaatleja kaugusele. See lähenemine sarnaneb suuresti käesolevas töös realiseeritud optimeeringuga, kuid erineb selle poolest, et võtab arvesse ka oksa raadiust ning ei kaota ära kaugema vaatleja korral väiksemaid oksi.

4.2. Võrdlus

Põhiline erinevus puude vormimise algoritmi ja L-süsteemidel põhineva algoritmi vahel on kasutamise intuitiivsus. Ka Paul Kruszewski artiklis on mainitud, et kasutatavus ja arusaadav puu parametriseerimine olid loodud algoritmi eesmärkideks. Vormimise algoritmil on lisas ära toodud kuus ühe puu kohta käivat üldist parameetrit ja kakskümmend puu iga taseme kohta käivat parameetrit. Väljatoodud parameetrid on määravad ära muu hulgas okste piirkujundi pikkust ja laiust, männaste vahelist pikkust, puutüve raadiust. Sellised parameetrid ütlevad intuitiivsemalt kasutajale ära, mille põhjal genereeritud puu tekkis, ja võimaldavad uut genereeritavat puud lihtsamini modifitseerida.

Ka L-süsteemil põhinevale algoritmile saab palju nendest parameetritest analoogselt realiseerida. Näiteks praeguses realisatsioonis on oksa tüvikoonuse põhjaraadius sõltuv oksast lähtuva kõige pikema ahela lülide arvust, kuid mõnedes rakendustes määratakse selle muutmiseks L-süsteemi tähemärgid. [8]

Probleem tekib aga kasutatavuses, sest soovitava puu genereerimiseks tuleb aru saada

formaalsest grammatikast ja selle tööpõhimõttest, et lisada või modifitseerida produktsioone, millest tulemus interpreteeritakse. On mõnevõrra intuitiivsem määrata oksa pikkus ja iga sellest eralduvate alam-oksade vahele jäävad pikkused, kui kirjutada mitmetele oksadele rakenduv tõenäosuslik reegel, mille alusel genereeritakse hargnemisi.

Seetõttu on kunstilise väärtusega puude genereerimise seisukohast lihtsam ja mõistlikum puude vormimise algoritm. L-süsteemide kasuks räägib aga süstematiseeritud puu kasvatamise võimalus. Ka L-süsteemide ajalugu vaadates selgub, et botaanikust Lindenmayer soovis nende abil modelleerida looduses esinevaid organisme, kellel on enesega sarnasuse omadus. Mitte aga võimaldada kunstnikele intuitiivset viisi looduses esinevatele seaduspärasustele mitte vastavate puude loomiseks.

Lähtudes rakendusest leitakse, et L-süsteemidel põhinev algoritm sobib mitmete lihtsamakoeliste puude genereerimiseks (vt näiteks Ekraanitõmmis 6), kui ei ole oluline puude täpne parametriseerimine. Samuti on L-süsteemid piisavalt võimsad, et genereerida ka muid enesega sarnaseid objekte peale puude. Kruszewski puude vormimise algoritm võimaldab aga kindla ja intuitiivse parametrisatsiooni alusel genereerida sobilikke puid. L-süsteemidel põhineva algoritmi üks võimalus on veel teisendada genereeritud puu mõne käsitsi modelleerimise programmi poolt loetavasse formaati, et kunstnik saaks seal loodud puud oma äranägemise järgi muuta. Sarnaselt on näiteks loodud vabavaralise 3D-modelleerimise tarkvara Blender jaoks L-süsteemide põhjal puude genereerimist võimaldavaid pistikprogramme. [9]

5. Kokkuvõte

Käesolev töö pakkus kirjeldavat ülevaadet kolmemõõtmeliste objektide protseduurilise genereerimise valdkonda puude modelleerimise ülesandest lähtuvalt. Realiseeriti ja kirjeldati tänapäeval levinud võimsate tehnoloogiatega raamistikku, mis tarkvara arhitektuuri seisukohast pakkus võimalust vastava domeeni ülesannete modulaarseks lahendamiseks. Töö raames näidati, kuidas programmeerimiskeelega C++ on võimalik kasutada mänguprogrammeerimise teeki Allegro graafilise kasutajaliidese programmeerimiseks, siduda rakendusega kolmemõõtmelise graafika teek OpenGL ning kasutades objekt-orienteeritud programmeerimise paradigma lähenemist lahendada vaadeldav ülesanne.

Tutvustati Lindenmayeri süsteemi ja selle stohhastilist varianti, mis töö raames ka realiseeriti. Seejärel näidati rakendatud grammatikaid ning nendega genereeritud puid. Optimeerimise peatükis tuldi uuesti tagasi kolmemõõtmelise keskkonna põhimõtete juurde ning seletati, kuidas nendest lähtuvalt on võimalik rakendust optimeerida.

Viimases peatükis on kirjeldatud Paul Kruszewski puude vormimise algoritmi ning vaadeldud, kuidas see erineb realiseeritud süsteemist nii põhimõtete kui ka kasutatavuse poolest. Sealjuures on mainitud ka mõningaid sarnasusi koostatud rakendusega ning aspekte, mille põhjal võib saada olemasolevat algoritmi veel täiustada.

Tehtud töö annab ülevaate kolmemõõtmelise keskkonnaga seotud ülesande põhilisest ülesehitusest. Samuti peaks töö pakkuma otsest seost teoreetilise formaalse grammatika ja praktilise väljundi vahel.

Rakenduse eeliseks lihtsamate L-süsteemi generaatorite ees, nagu JavaScriptiga tehtud Nolar Carrolli [10] või Kevin Roasti [6] rakendused, on L-süsteemide interpreteerimine kolme-mõõtmelisteks puudeks, mainitud autorite interpretaatorid suudavad joonistada ainult kahemõõtmelisi struktuure. Teisest küljest pakub loodud rakendus lihtsat võimalust katsetada L-süsteemide või näha mõne muu puugenereerimise algoritmi tulemusi ilma, et peaks valdama järsu õpikõveraga 3D-modelleerimistarkvara nagu SpeedTree [11] või Xfrog [12]. Viimaste kasuks räägib küll võimalus genereeritud puid käsitsi sobivaks muuta ja salvestada neid tuntud formaatidesse, ent mõlema puhul on tegu äriklassi tarkvaraga, mis muudab kättesaadavuse raskeks.

Võrreldes teiste analoogsete projektidega võib välja tuua Ismail Habib'i projekti [13],

milles puudub stohhastilise L-süsteemi toe võimalus. Natuke suurem projekt eelmainitust on Hora [8], mis osutus mõnedes kohtades ka käesoleva töö eeskujuks. Seal L-süsteeme kasutavas puude- ja maastikugeneraatoris tekivad aga puu segmentide vahele tühimikud ja kõige viimased segmendid jäävad lahtise otsaga. Käesolevas realisatsioonis selliseid olukordi ei juhtu.

Töö koostamine oli minu hinnangul huvitav, sest seni ei olnud OpenGL-iga kokku puutunud ning töö raames õppisin väga palju kolmemõõtmeliste objektide kujutamise kohta. C++'i ja Allegroga oli küll kokkupuude olemas, kuid puudus suuremamahulise projekti realiseerimise kogemus, mida antud tööst ootasin ja sain. Kahjuks ei jäänud töö raames aega rakendada ka Kruszewski puude vormimise algoritmi, mis esialgu oli plaanis. Kõige rohkem aega võttis OpenGL-i ja kolmemõõtmelise ruumi matemaatika õppimine ning realiseerimine.

Edasi on võimalik realiseeritud raamistiku peale realiseerida Kruszewski puu vormimise või mõni muu puu genereerimise algoritm. Samuti on võimalik erinevate L-süsteemidega puid genereerida ja proovida rakendada teistsuguseid produktsioone. Ühe edasiarendusena on võimalik koostada ka puudele lehtede genereerimise algoritm või kasutada tekstuure, et muuta puid efektsemateks. Saab edasi arendada ka optimeeringuid ning viia objektide joonistamine OpenGL-i nimistute pealt töös mainitud tipumassiivide või tipu vahemälu objektide (VBO) kasutamisele. Ka kasutajaliidese osas on võimalik teha vahetuid edasi-arendusi, näiteks realiseerida võimalus kasutajaliidese abil puude selekteerimiseks ja nende info (näiteks L-süsteemi sõne, mille pealt puu genereeriti) vaatamiseks.

Tänan enda juhendajat Konstantin Tretjakovi ja kaastudengit Timo Kallastet, kes aitasid OpenGL-i mõistete ja rakendusviisidega. Veel tänan ka juhendajat Sven Lauri ja tema kirjutamisrühma eesotsas Janno Jõgevaga, kes aitasid töö vormistamise, sisulise ja organisatoorsete küsimuste osas. Samuti tänan magistritudengeid Tõnis Tobret ja Henri Lakk'i, kes pakkusid abi süsteemiarhitektuuri ja algoritmidega tekkinud probleemide korral.

Procedural Tree Generation

Bachelor's thesis (6 EAP)

Raimond-Hendrik Tunnel

Summary

Thesis introduces the problem of procedurally generating trees in computer graphics, explaining the need for such an approach and what some of the possible solutions would be. Thesis also explains lot of the basic functionality in OpenGL, showing an example how to create and render a three-dimensional world using C++, Allegro and OpenGL.

First chapter shows how we created a framework using the fore mentioned technologies and what one should take into account when creating such a system. It is also explained what do Allegro and OpenGL libraries actually give for solving a problem in a three-dimensional virtual environment.

Third chapter concentrates on the most traditional way of generating trees – using a variant of formal grammar called Lindenmayer system. We give a strict definition of L-system and show how it can be implemented to assemble and render trees. In addition, we bring out some of the problems that may occur when rendering three-dimensional trees and offer possible solutions to these problems while analyzing basic concepts of OpenGL to show why they arose in the first place.

In the fourth chapter we introduce another approach described by Paul Kruszewski in his article *An algorithm for sculpting trees*. The algorithm is described in some detail and we have explained what kind of considerations Kruszewski has taken into account when generating trees. There is also a comparison between the L-system and Kruszewski approaches from the user's perspective, explaining the intuitiveness of both algorithms.

For conclusion we compare briefly our implementation of L-system based tree generation to couple of other projects using the same approach, and also to business-class software that allow for direct modelling of generated trees. We also state how the implementation can be improved to support more complex trees, generate leaves or assemble trees based on some other algorithms.

Viited

[1] Matthew Leverton, Allegro.cc, 2012, http://www.allegro.cc/ (14. 05. 2012).

[2] M. Woo, J. Neider, T. Davis, OpenGL Programming Guide, 1997, http://www.glprogramming.com/red/ (14. 05. 2012).

[3] M. Woo, J. Neider, T. Davis, OpenGL Programming Guide: OpenGL as a State Machine, 1997, http://www.glprogramming.com/red/chapter01.html#name4 (14. 05. 2012).

[4] Gabriela Ochoa, An Introduction to Lindenmayer Systems, 1998, http://www.biologie.uni-hamburg.de/b-online/e28_3/lsys.html (14. 05. 2012).

[5] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, The Algorithmic Beauty of Plants, 1990, Springer-Verlag.

[6] Kevin Roast, L-systems Turtle Graphics Renderer, 2009, http://www.kevs3d.co.uk/dev/lsystems/ (14. 05. 2012).

[7] Paul Kruszewski, An algorithm for sculpting trees, Pergamon, 1999.

[8] George P. Stathis, Procedural Landscape and Tree Generation: "Hora", 2002, http://george-stathis.com/hora.html (14. 05. 2012).

[9] Liquidweb, L-system, 2001, http://lsystem.liquidweb.co.nz/ (14. 05. 2012).

[10] Nolar Carroll, L-system, 2010, http://nolandc.com/sandbox/fractals/ (14. 05. 2012).

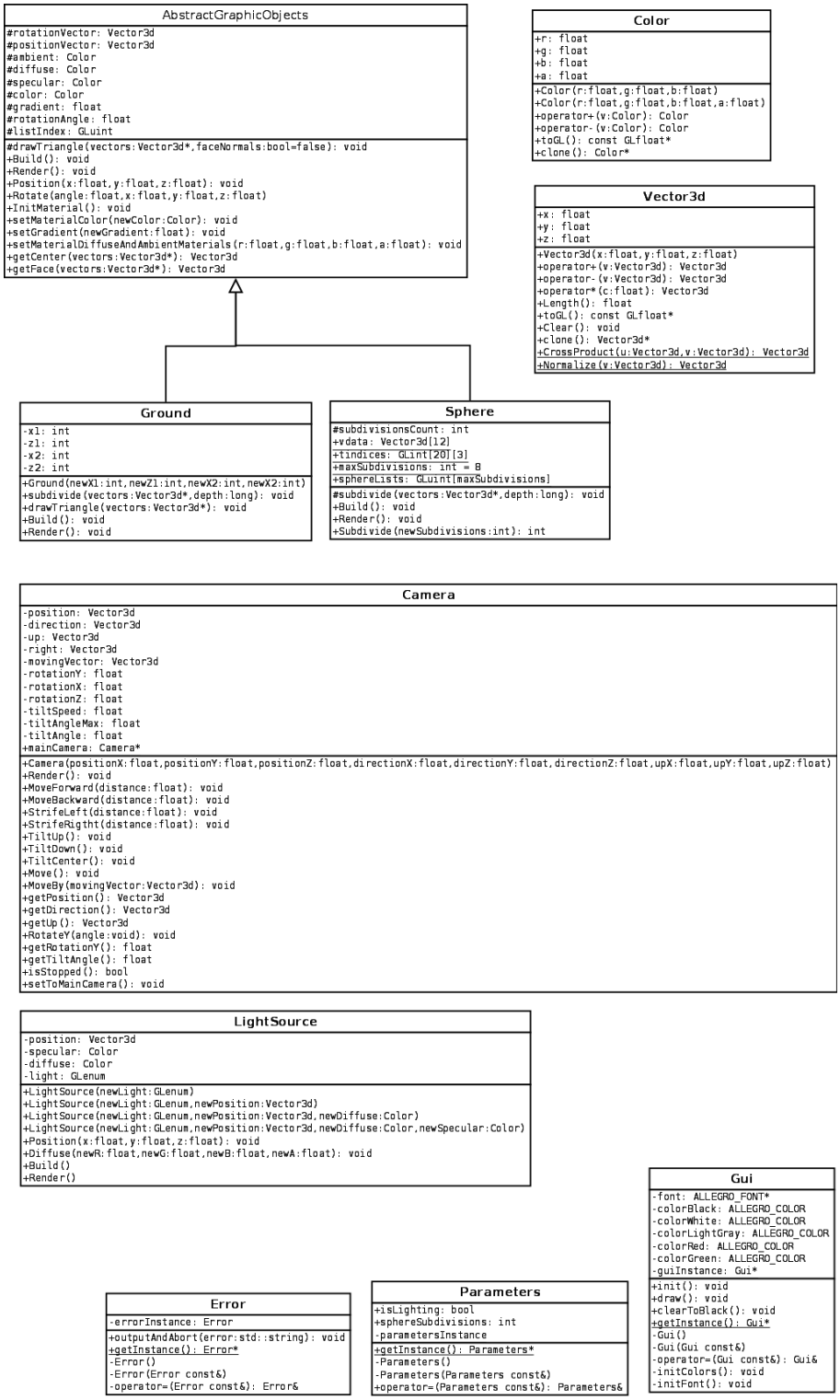
[11] IDV Inc., SpeedTree, 2012, http://www.speedtree.com/ (14. 05. 2012).

[12] Xfrog Inc., Xfrog, 2012, http://xfrog.com/ (14. 05. 2012).

[13] Ismail Habib, Modelling a Tree using L-system, 2008, http://www.geekyblogger.com/2008/04/tree-and-l-system.html (14. 05. 2012).

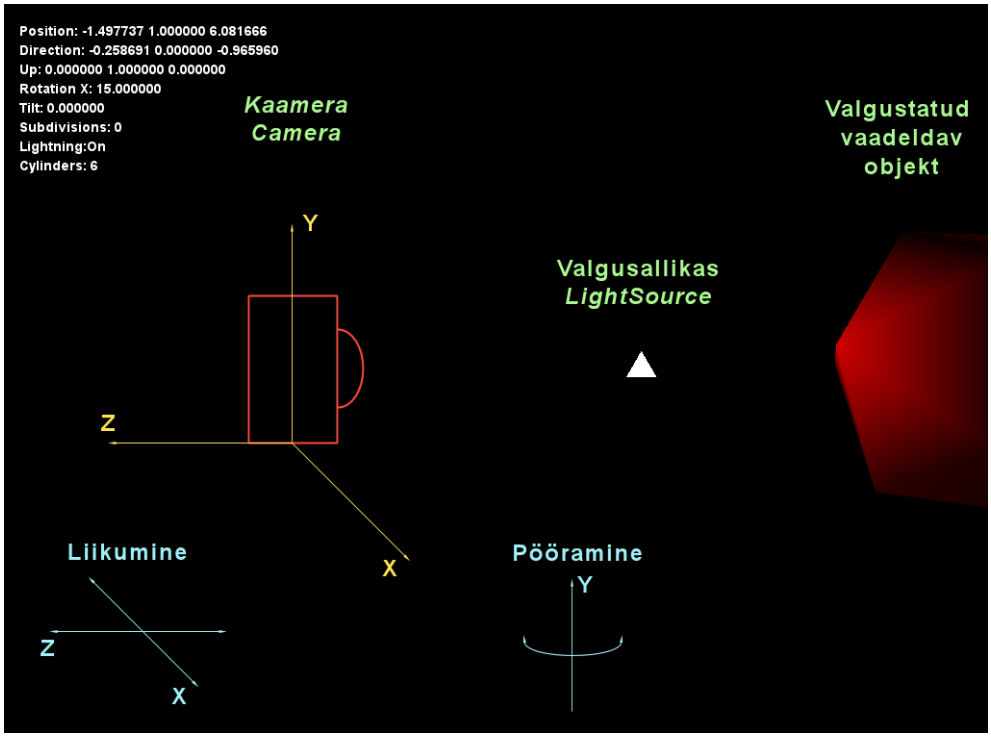
Lisad

Lisa 1

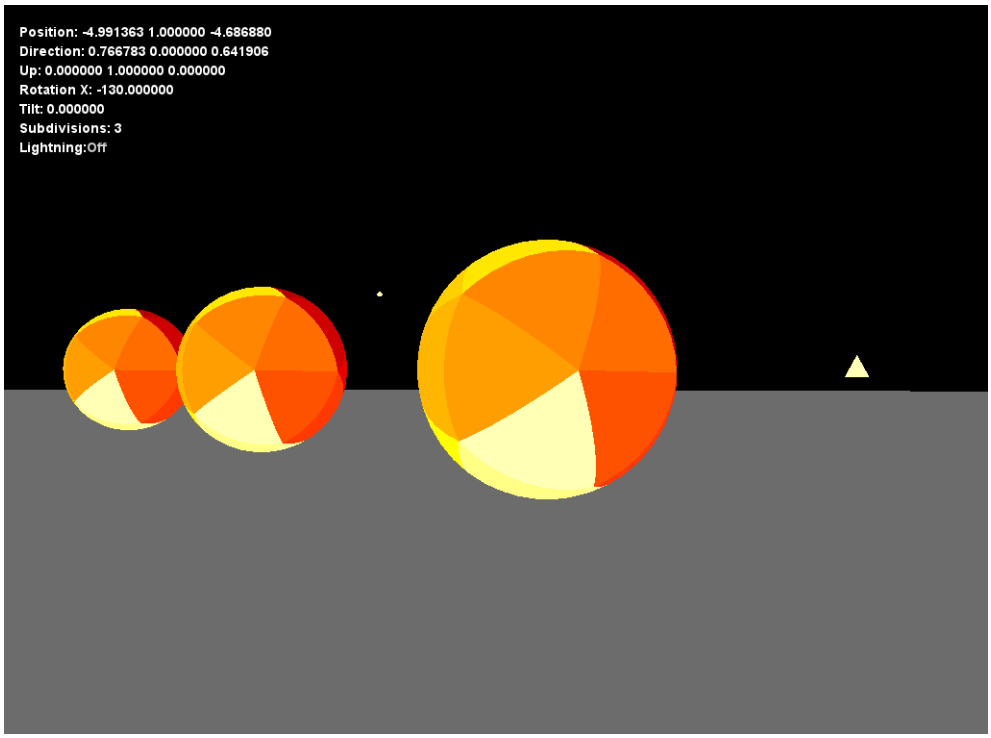


Graafik 1: Klassidiagramm

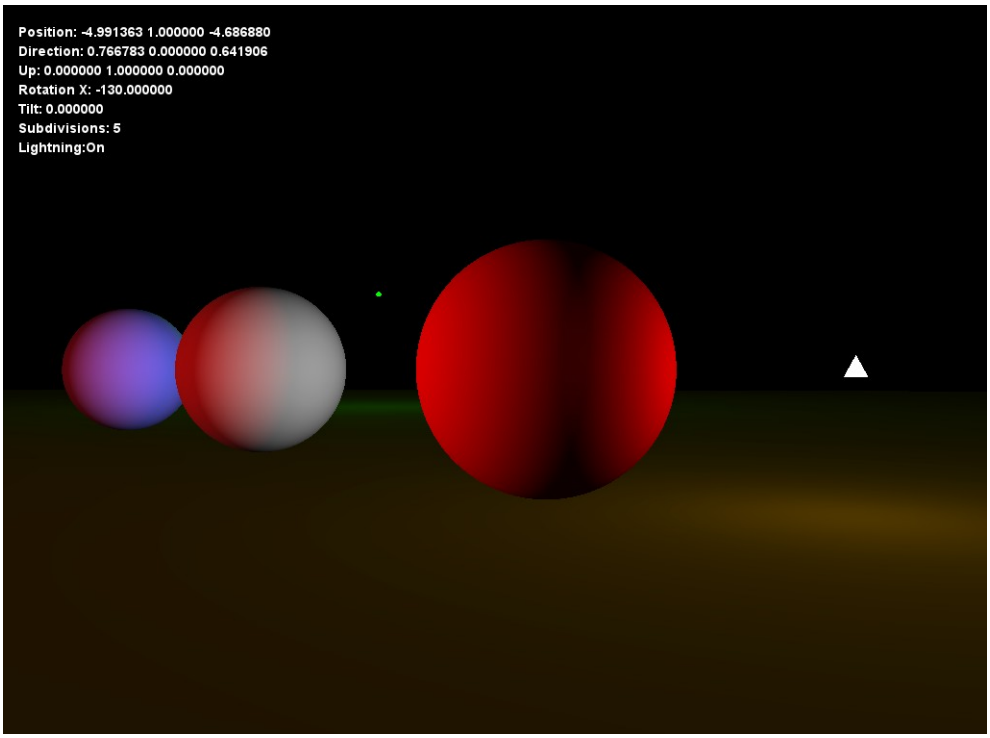
Lisa 2



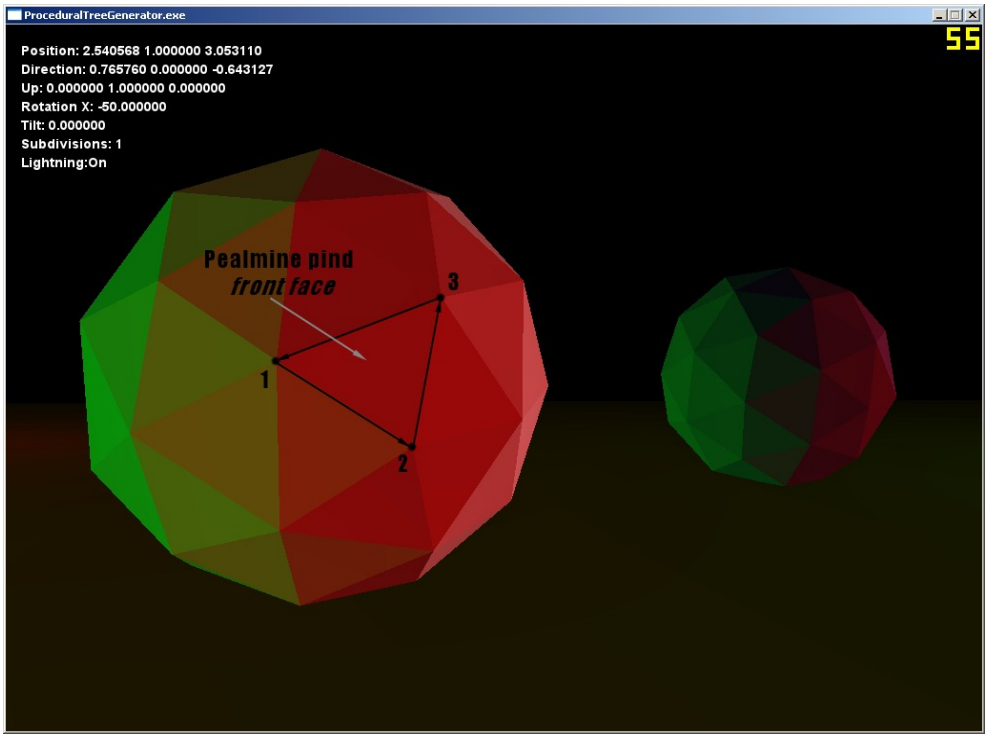
Pilt 4: Kaamera liikumisvabadust selgitav joonis



Ekraanitõmmis 1: Vaade rakendusse - ilma valgustuseta



Ekraanitõmmis 2: Vaade rakendusse - valgustus sisselülitatud. Valgusallikad on esiplaanil olev valge kolmnurk ja tagaplaanil olev roheline kolmnurk (kiirgavad vastavaid värve). Vasakult kiirgab punane valgusallikas. Sfäärid on vaataja poolt värvidega punane, hall, helesinine.

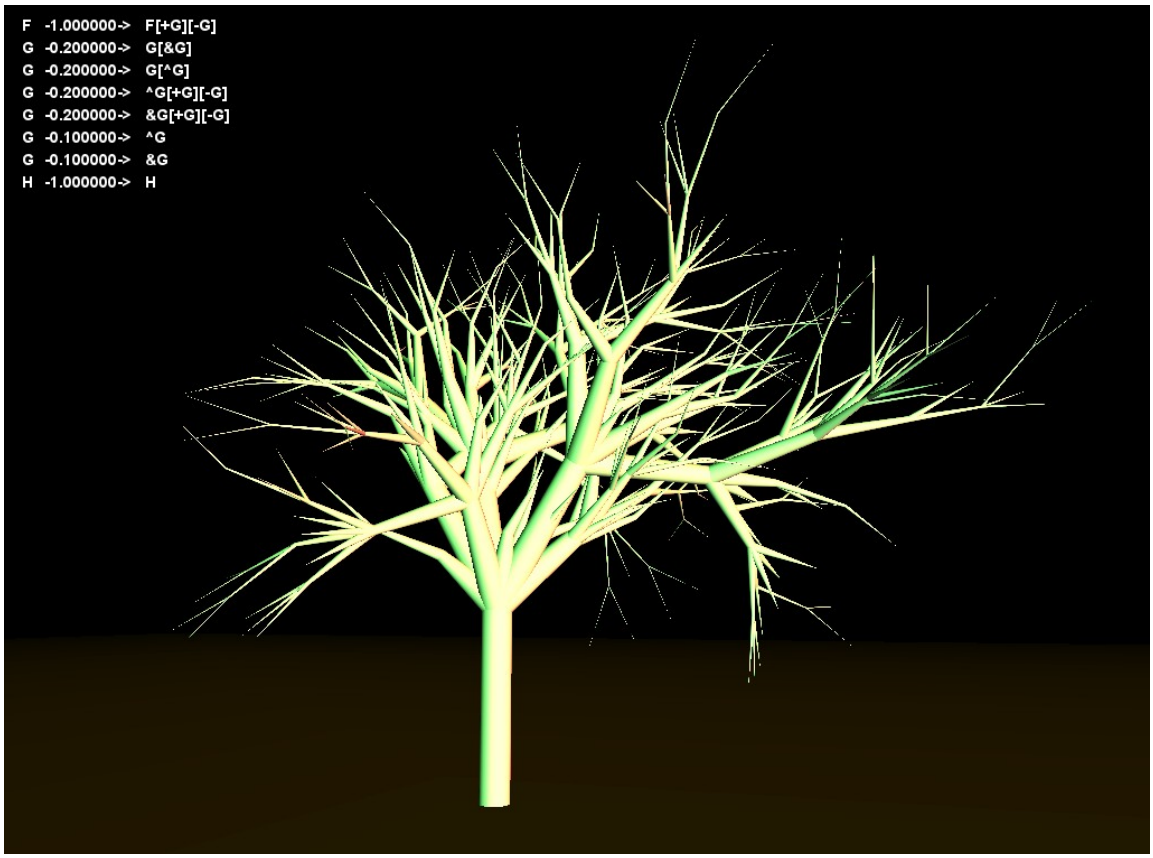


Ekraanitõmmis 3: Rakenduses on normaalid tõmmatud kolmnurkade pindadele mitte otspunktidele, et pinnad oleksid üksteisest eristuvamad. Ühe kolmnurga tipud defineeritakse järjekorras: 1, 2, 3.

Lisa 3

Ekraanitõmmis 5: Puu produktsioonidega
LProduction first1 = LProduction("F", "F[^F[+F][-F]]", 0.5);
LProduction second1 = LProduction("F", "F[&F[+F][-F]]", 0.5);
ja iteratsioonide arvuga 3 aksioomist F.

Ekraanitõmmis 6: Puu produktsioonidega
LProduction first1 = LProduction("F", "H[+G][-G]", 1);
LProduction second1 = LProduction("G", "H[&G]", 0.3);
LProduction second2 = LProduction("G", "H[^G]", 0.3);
LProduction second3 = LProduction("G", "^H[+G][-G]", 0.2);
LProduction second4 = LProduction("G", "&H[+G][-G]", 0.2);
LProduction third1 = LProduction("H", "H", 0.9);
LProduction third2 = LProduction("H", "H[&F]", 0.05);
LProduction third3 = LProduction("H", "H[^F]", 0.05);
ja iteratsioonide arvuga 6 aksioomist F.



Ekraanitõmmis 7: Puu produktsioonidega:
LProduction first1 = LProduction("F", "F[+G][-G]", 1);
LProduction second1 = LProduction("G", "G[&G]", 0.2);
LProduction second2 = LProduction("G", "G[^G]", 0.2);
LProduction second3 = LProduction("G", "^G[+G][-G]", 0.2);
LProduction second4 = LProduction("G", "&G[+G][-G]", 0.2);
LProduction second5 = LProduction("G", "^G", 0.1);
LProduction second6 = LProduction("G", "&G", 0.1);
ja iteratsioonide arvuga 6 aksioomist F.

Lisa 4

SVN repositoorium

http://subversion.assembla.com/svn/procedural-tree-generation/trunk/

Tarkvara on testitud järgneva spetsifikatsiooniga arvutil:

CPU: Intel Core 2 Duo E6400, 2.13Ghz

RAM: 3GB

Video: ATI Radeon HD 5570, 1GB

OpenGL versioon 4.1.10524