SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 689

PROCEDURALNO GENERIRANJE GRAFIČKIH OBJEKATA

Marko Vrljičak

Zagreb, lipanj 2014.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 7. ožujka 2014.

DIPLOMSKI ZADATAK br. 689

Pristupnik: **Marko Vrljičak (0036449668)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Proceduralno generiranje grafičkih objekata**

Opis zadatka:

Proučiti postupke koji omogućuju proceduralno generiranje grafičkih sadržaja koji se koriste u izgradnji virtualnih okruženja. Razmotriti zahtjeve tih postupaka na vremensku i memorijsku komponentu pri izračunu. Razmotriti povezanost ovih postupaka s genetskim algoritmom. Ostvariti programsku implementaciju koja omogućuje prikaz razrađenog modela. Na različitim primjerima prikazati ostvarene rezultate. Diskutirati utjecaj parametara. Načiniti ocjenu implementiranog algoritama i ostvarenih rezultata.
Izraditi odgovarajući programski proizvod. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku: 14. ožujka 2014.
Rok za predaju rada: 30. lipnja 2014.

Mentor:
Prof.dr.sc. Željka Mihajlović

Djelovođa:
Doc.dr.sc. Tomislav Hrkać

Predsjednik odbora za diplomski rad profila:
Prof.dr.sc. Siniša Srbljić

Sadržaj

Uvod

Proceduralno generiranje podrazumijeva stvaranje sadržaja kroz nasumične ili pseudo-nasumične procese koji rezultiraju velikim brojem mogućih rezultata. Ovaj način stvaranja sadržaja popularan je u računalnoj grafici, ponajviše igrama i najčešće se odnosi na stvaranje terena, objekata, tekstura i zvuka. Najčešće primjene ovog načina stvaranja sadržaja u računalnim igrama su vjerojatno uređivanje virtualnog svijeta (engl. *level design*) i stvaranje dinamičkih tekstura. Ideja iza proceduralnog generiranja je korištenje sadržaja stvorenog tijekom izvođenja programa umjesto spremanja gotovog sadržaja na disk kako bi se koristio prilikom izvođenja – proceduralno generiranje koristi procesorsku moć za generiranje sadržaja koji bi se inače učitavao iz memorije.

Ovaj rad obrađuje proceduralno generiranje grafičkih objekata, koji se kasnije mogu razmjestiti po virtualnoj sceni daljnjim korištenjem sličnih algoritama. Sadržaj je moguće u potpunosti proceduralno generirati stvaranjem objekata i tekstura kroz kod, no češće se koristi neka od metoda koje stvaraju tek dio sadržaja na taj način ili samo povezuju sadržaj proceduralno. U nastavku biti će opisano nekoliko metoda i primjera iz ovog širokog raspona, od kojih će neki biti detaljnije razrađeni i implementirani.

Evolucijski algoritmi su područje računarske znanosti koje se koristi za dostizanje optimuma neke funkcije koristeći djelomično nasumične i samostalne jedinice. Zbog popularnosti, ali i potencijala ovakvih metoda, obrađeno je i korištenje nekih evolucijskih algoritama prilikom generiranja proceduralnog sadržaja, te usporedba takvih metoda s klasičnima.

Popularnost ovakvih metoda u računalnim igrama samo raste, što se može vidjeti i iz širenja zasebnih žanrova koji se zasnivaju na proceduralnom generiranju sadržaja – takozvane *roguelikes* su igre nalik na igru *Rogue* iz 1980. čija je osnovna karakteristika različitost sadržaja pri svakom pokretanju, a zbog varijacija unutar samog žanra nastaju brojni moderni podžanrovi. Postoje i mnogi programi koji služe za stvaranje cijelih svjetova (na primjer *Terragen*), a brojne su i primjene u filmskoj industriji. Zbog povezanosti s fraktalima ovo područje je iznimno zanimljivo u računalnoj grafici, djelomično zbog toga što se za izvođenje ovih metoda često koristi grafički procesor [1].

Svi primjeri implementirani su u Unity3D programskom pogonu (engl. *engine*) i C# programskom jeziku.

1. Proceduralno generiranje sadržaja

Postoje brojne taksonomije koje pokušavaju opisati podjelu ovog područja na primjene u igrama. Jedna od zanimljivijih [2] opisuje proceduralno generiranje sadržaja kroz slijedećih sedam kategorija:

1. Dinamičko stvaranje razina igre tijekom igranja (engl. *runtime random level generation)*
2. Uređivanje virtualnog svijeta tijekom koraka dizajniranja igre (engl. *design of level content*)
3. Dinamičko stvaranje svijeta kroz igru (engl. *dynamic world generation*)
4. Stvaranje entiteta unutar igre (engl. *instancing of ingame entities*)
5. Sadržaj posredno stvoren od strane korisnika (engl. *user mediated content*)
6. Dinamički sustavi upravljanja (engl. *dynamic systems*)
7. Proceduralne zagonetke i stvaranje zapleta (engl. *procedural puzzles and plot generation*)

Kroz ovaj rad ponajviše će biti obrađena točka 4, uz doticanje točaka 1, 3 i 5. Točka 2 u nekim taksonomijama niti ne spada u ovo područje zbog toga što se jedina nasumičnost obrađuje tijekom izrade igre – igranje je potpuno deterministično. Točke 6 i 7 ne spadaju u potpunosti pod stvaranje grafičkog sadržaja, pa ih neću detaljnije obraditi.

Od svojih početaka, jedna od niti vodilja proceduralne grane generiranja sadržaja – od teksta, preko zvuka do objekata i cijelih svjetova – jest ušteda prostora pod cijenu što manje količine vremena. U mrežnim igrama ovo znači da se sadržaj može generirati na računalu servera i prikazivati kod klijenata umjesto čuvanja istoga na klijentovoj strani u obliku zapisa na čvrstom disku [3]. Tu se koriste razni mehanizmi [4], od jednostavnih funkcija aproksimacije do kompliciranih gramatika i automata [5] kako bi se simulirala nasumičnost, smanjila potreba za prostorom i zadržala brzina izvođenja. Za demonstraciju moći procedura kao oruđa često se koriste generatori terena koji mogu stvoriti vrlo različite, ali potpune svjetove promjenom samo jednog parametra. Pravilna i prirodna

promjena parametara kroz vrijeme je zato ključna za dobivanje uvjerljivog i zanimljivog rezultata [6].

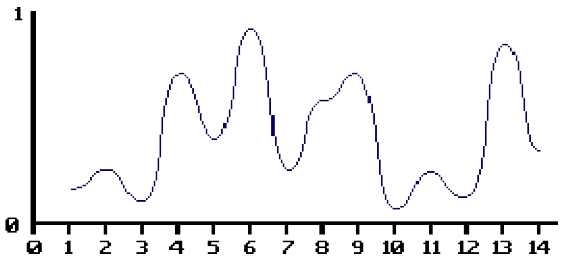
1.1. Stvaranje nasumičnosti

Za stvaranje prostranih proceduralnih svjetova potrebno je prvo razmotriti njegove najmanje djelove. Kako se ova cijela grana zasniva na nasumičnom ili barem prividno nasumičnom odabiru između poželjnih rezultata, potrebno je definirati načine na koje se takav odabir može napraviti.

Standardna nasumična funkcija s uniformnom ili normalnom razdiobom je često preoštra da bi izgledala prirodno, iako je osnova većine nedeterminističkih algoritama (pa tako i onih korištenih u proceduralnoj grafici). Stoga se u primjenama u kojima je potrebno generirati i najmanje detalje da izgledaju prirodno koristi neki od šumova, od kojih su vjerojatno najpoznatiji Perlinov i simpleks šum. Simpleks šum je ubrzanje originalnog Perlinovog algoritma uz mogućnost bolje paralelizacije i jednostavne implementacije na sklopovskoj opremi, no Perlinov šum predstavlja početak stvaranja gotovo svog proceduralnog sadržaja te je intuitivniji, pa ću, usprkos brojnim varijantama, detaljnije opisati samo originalni algoritam.

1.1.1. Perlinov šum

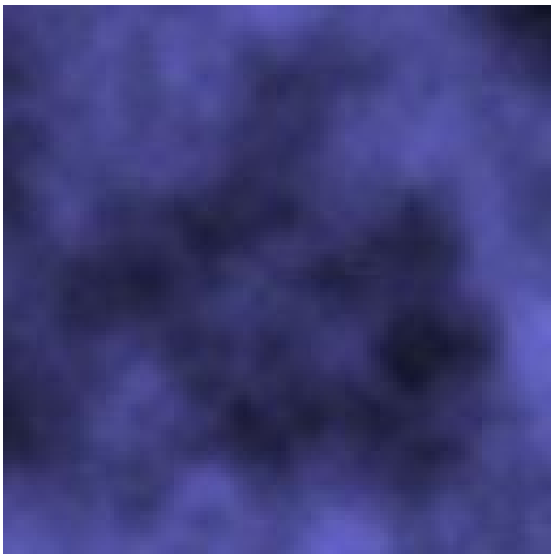
Primjeri koji opisuju stvarni svijet kroz fraktale dobro opisuju i način stvaranja Perlinovog šuma [7]. Čest primjer je obris lanca planina. Na velikoj udaljenosti prepoznaju se tek grubi obrisi planina, ali kako se približavamo vidimo da slične obrise imaju i brda, kamenje i kamenčići. Perlinov šum se stvara na sličan način – zbrajanjem šuma različitih veličina. Funkcije šuma su vrlo jednostavni generatori nasumičnih brojeva, ali za njih je bitno napomenuti da za isti ulaz uvijek daju isti izlaz, neovisno o broju dimenzija ulaza. Prirodni šum moguće je dobiti interpoliranjem nasumičnih vrijednosti na nekom razmaku tako da funkcija šuma izgleda glatko.



Slika 1 – Interpolirani jednodimenzionalni šum [7]

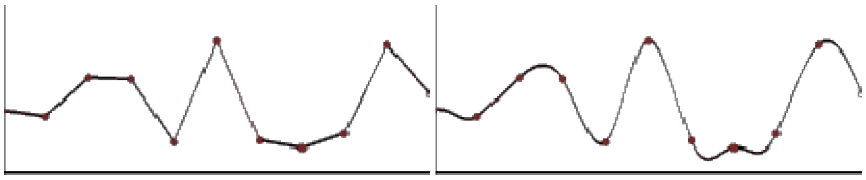
Parametri šuma su amplituda i frekvencija – amplituda predstavlja razliku između minimalne i maksimalne vrijednosti, a frekvencija učestalost nasumičnih vrijednosti koje se interpoliraju. Stoga ćemo povećanjem amplitude i smanjenjem frekvencije dobiti „veći“ šum (niža oktava), a smanjenjem amplitude i povećanjem frekvencije „manji“ šum (viša oktava). Perlinov šum je zbroj šumova različitih oktava, što rezultira šumom koji je prepun manjih, srednjih i većih varijacija – ovisno o broju oktava zbrojenih u šum.

Slika 2 – Jednodimenzionalni Perlinov šum



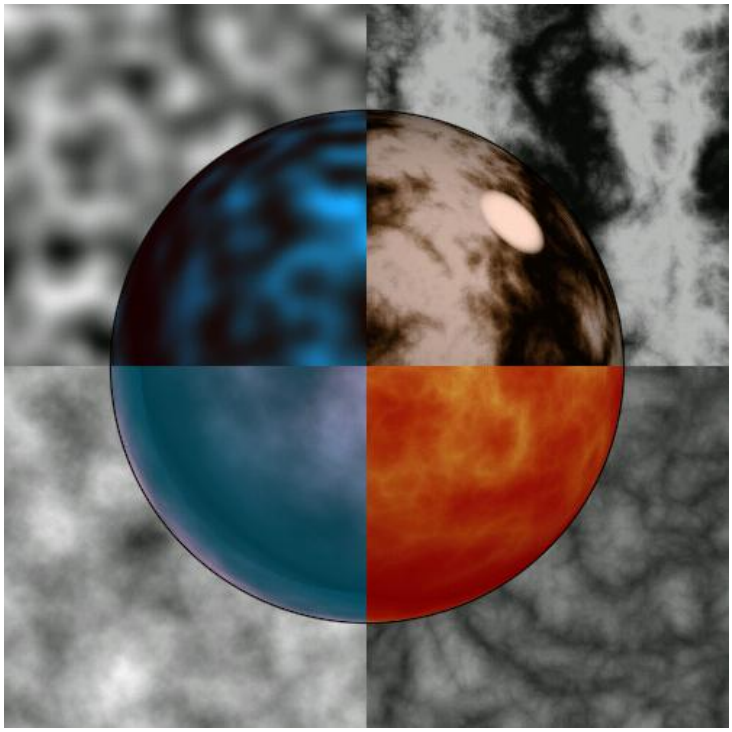
Slika 3 – Dvodimenzionalni Perlinov šum [7]

Jedan važan detalj šuma je način interpolacije između pseudo-nasumičnih vrijednosti. Linearna interpolacija je brza i koristi se u slučajevima gdje se šum stvara vrlo često u stvarnom vremenu, dok je kubna interpolacija vrlo precizna i glatka, ali spora. Bilinearna i trilinearna interpolacija koriste se za više dimenzije – naime, dok jednodimenzionalni šum nastaje kao interpolacija dvaju pseudo-nasumičnih brojeva, dvodimenzionalni šum koristi četiri takva broja, a trodimenzionalni osam. Pseudo-nasumični brojevi dobivaju se iz prethodno izračunatih tablica koje služe za optimizaciju brzine ili iz generatora takvih brojeva, vjerojatno najpoznatiji od kojih je Mersenne twister [1]. Osim interpolacije, šum se često izglađuje dodatnom funkcijom koja u obzir uzima samo neposredne susjede.



Slika 4 – Linearna (lijevo) i kubna (desno) interpolacija [7]

Primjene Perlinovog šuma su brojne, neovisno o broju dimenzija u kojima se koristi. Jednodimenzionalni Perlinov šum može zamijeniti nasumični generator brojeva, a koristi se za male pomake, poput pomicanja cijelog ili dijelova virtualnog lika (jer stajanje savršeno mirno izgleda neprirodno), simuliranje nesavršenosti ljudskog crtanja i pisanja (jer linije crtane rukom nikad nisu savršeno ravne) i mnogih drugih primjena.



Slika 5 – Perlinov šum proveden kroz razne funkcije [8]

Dvodimenzionalni Perlinov šum služi za generiranje krajolika koji se potencijalno proteže unedogled do najmanjih detalja, dvodimenzionalnih oblaka i raznih tekstura koje mogu narasti vrlo velike prije nego što se uoči ikakvo ponavljanje šuma [8, 9].

Trodimenzionalne i četverodimenzionalne primjene Perlinovog šuma najčešće uključuju trodimenzionalne objekte ili animaciju dvodimenzionalnih ili trodimenzionalnih objekata – ona dodatna dimenzija u ovom slučaju predstavlja vrijeme.

Iznimno korisno svojstvo šuma jest da će za isti slijed ulaza (na primjer, vremena od početka simulacije) dati isti slijed izlaza, a generatora pseudo-nasumičnih brojeva da uz iste početne parametre daju isti slijed. Ovo omogućuje stvaranje velikih količina podataka promjenom samo jednog parametra koji mijenja ulaz u funkciju generatora brojeva.

Sam šum posjeduje neke pravilnosti, poput redovitog vraćanja svake oktave u nulu [9]. Na osnovi tih pravilnosti, korištenjem vrijednosti šuma iz jedne ili nekoliko poslijednjih iteracija moguće je pravilno stvarati objekte u beskonačnom nizu, na primjer relativno ravnomjerno raspoređeni objekti u dvodimenzionalnoj igri.

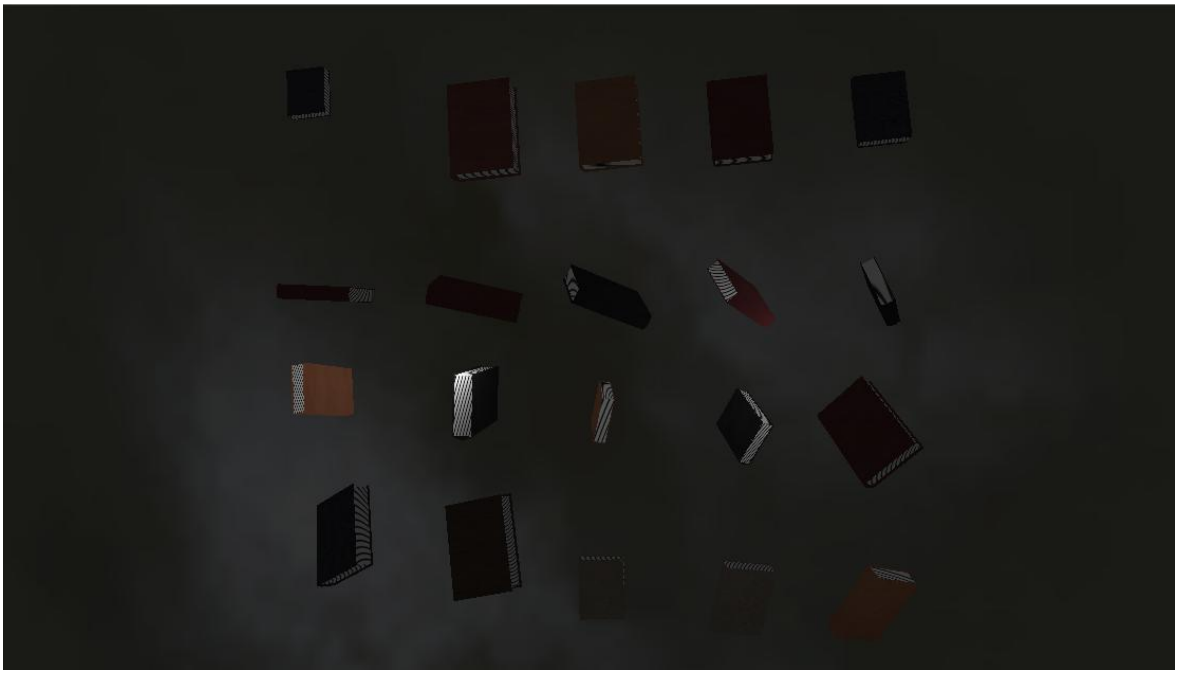
Slika 6 – Generiranje prirodnog pravilnog ponavljanja iz šuma

Složenost izračuna Perlinovog šuma je $O(2^{n-1})$ po jedinici šuma, gdje je n broj dimenzija, što brzo raste i već nakon nekoliko dimenzija prestaje biti izračunljivo – no, postoje brojne optimizacije (uključujući simpleks šum, čija složenost je $O(n^2)$) koje omogućuju smanjenje ove složenosti. Jedna takva optimizacija zasniva se na tome da je u jednom trenutku moguće vidjeti tek ograničeni dio prostora šuma, a ukoliko bi pokušali vidjeti više, ta razina detalja šuma bila bi toliko udaljena da se ne bi niti vidjela (odnosno, izgledala bi ravno). Tako je moguće ponavljati šum nakon određene, relativno bliske točke, uz interpolaciju između istih područja ili korištenje modifikacije koja čini šum ponovljivim u svim smjerovima bez prekida [8, 9, 10].

Kroz ovaj rad koristi se nekoliko varijanti Perlinovog šuma s različitim parametrima, najčešće kao generatori pravilnih, nasumičnih, jednodimenzionalnih slijedova vrijednosti, ali gotovo uvijek kao dio složenijeg algoritma.

1.2. Potpuno proceduralno generiranje objekata

Vjerojatno najintuitivniji način stvaranja proceduralnih virtualnih objekata jest zasebnim definiranjem položaja svakog vrha modela (engl. *vertex*), svakog poligona (engl. *triangle*), preslikavanja tekstura (engl. *texture mapping*) i određivanjem svih detalja takvog objekta direktno kroz kod. Ovakvo stvaranje se brzo izvodi i jednostavno je za razumjeti te su rezultati obično predvidljivi, ali njegova upotreba je ograničena (program koji stvara kamenje različitih oblika neće se jednostavno moći prenamijeniti za stvaranje nekog drugog tipa objekata) i programiranje složenijih objekata može biti vrlo naporno. Dodavanje novih mogućnosti u ovakav proces nije jednostavno, te se ovaj način modeliranja rijetko koristi – osim za rijetke primjene neučinkovito je u usporedbi s učitavanjem spremljenog modela i kasnije modifikacije istoga. Moguće je koristiti ovakvo generiranje objekata kao predprocesiranje za neki drugi algoritam [11]. Ovdje je spomenuto isključivo kao demonstracija brzine proceduralnog generiranja potpunog modela s teksturama. Brzina stvaranja jedne knjige je u prosjeku 20 ms. Ovo, kao i sva preostala mjerenja u radu, izvršena su na 2.53 GHz Intel Dual Core procesoru istovremeno. Ideja iza takvog mjerenja bila je pokazati relativnu brzinu metoda koje su ovdje korištene i naglasiti da su same vrijednosti znatno lošije od onih koje se mogu dobiti optimizacijom.



Slika 7 – Generator knjiga

Druga primjena ovakvog načina stvaranja modela je sinteza modela. Taj način stvaranja zasniva se na zamjenjivosti dijelova između više sličnih modela istog tipa u svrhu

generiranja velikog broja različitih konačnih objekata (knjige možda jesu različite u detaljima, ali su izdaleka vrlo slične). Umjesto korištenja gotovih dijelova moguće je koristiti funkcije koje generiraju neki dio parametarski (time je moć upravljanja rezultatom znatno veća, a i prostor mogućih rješenja se povećava). Zbog toga što su dijelovi predmeta često vrlo jednostavni, oni se mogu generirati dijelom ili u potpunosti proceduralno. Na primjer, osnovni model stola sastoji se od nekog broja nogu i gornje površine. Sama površina stola može biti oblika spljoštenog valjka ili kvadra, a noge mogu biti napravljene od istih tih oblika, samo uz drugačije skaliranje dimenzija. Dodavanjem opcija poput proizvoljne veličine, broja nogu i dijelova, moguće je napraviti generator stolova koji koristi veliki broj parametara, što daje dojam velike raznolikosti. Ovi parametri često su određeni nasumično kroz neki raspon vrijednosti za koje generator daje prihvatljive rezultate. Zaključavanjem nekih opcija na zadanu vrijednost ili raspon dobivaju se napredne opcije u programima za automatsko generiranje sadržaja – neki skup raspona parametara mogao bi stvarati, na primjer, samo barokne stolove.

2. Primjena evolucijskih algoritama

Evolucijski algoritmi, kao dio evolucijskog računarstva, spadaju u područje umjetne inteligencije u računarskoj znanosti. To su algoritmi optimizacije inspirirani biološkom evolucijom - najčešće pojmovima selekcije, mutacije i križanja. Velika prednost ovakvih algoritama je u tome što su primjenjivi na širok spektar problema jer ne pretpostavljaju mnogo o samome problemu, dok su istovremeno otporni na zastajanje u lokalnim optimumima zbog mehanizama mutacije [12].

Genetski algoritam je najrasprostranjenija vrsta evolucijskih algoritama, koja imitira procese prirodne selekcije. Kroz algoritam, populacija rješenja provodi se kroz ocjenjivanja i modifikacije kako bi postigla zadovoljavajuće rješenje. Najčešći, osnovni primjer genetskog algoritma uključuje populaciju u kojoj se svakoj jedinki pridaje neka dobrota ovisno o njenoj blizini optimumu ciljne funkcije. Zatim se, ovisno o dobroti, odaberu dvije prikladne jedinke čija se rješenja pomiješaju i na kraju se novonastala jedinka nasumično promijeni. Svaka iteracija algoritma stoga sadrži slijedeće korake:

- Evaluacija: ocjenjivanje jedinki populacije prema ciljnoj funkciji ili pravilima
- Selekcija: odabir jedinki čije će se rješenje prenijeti u buduće iteracije
- Križanje: razmjena dijelova rješenja između odabranih jedinki populacije
- Mutacija: nasumična promjena dijelova rješenja u populaciji

Zbog načina selekcije, križanja i mutacije, ovaj proces moguće je ubrzati paralelnom obradom, no dobivanje konačnog, točnog rješenja ne mora biti prioritet ni način na koji se koriste genetski algoritmi u grafici (jedna od mana genetskog algoritma je upravo problem zaustavljanja, jer je u mnogim slučajevima teško ili nemoguće odrediti ciljnu vrijednost, pa je dobiveno rješenje samo bolje od svih drugih dobivenih kroz to izvođenje algoritma). Elitizam ili elitna selekcija je varijanta genetskog algoritma koja osigurava da će do nekoliko najboljih jedinki iz neke iteracije bez promjene dospjeti u slijedeće iteracije. Korištenjem elitizma pogreška koju algoritam koristi neće nikada rasti, pa najbolje rješenje polako konvergira. Možda najveća mana genetskog algoritma, njegova brzina, dijelom proizlazi iz iste postavke iz koje proizlazi njegova prednost – zbog malo pretpostavki o nekom problemu, algoritam nije prilagođen rješavanju tog specifičnog problema. Drugi dio

problema (i mogući razlog zašto se evolucijski algoritmi rijetko koriste u grafici) je u slaboj prilagodbi na kompliciranije i iznimno jednostavne probleme – zbog mehanizma mutacije za kompliciranе probleme vrijeme i prostor pretrage raste eksponencijalno, a za manje probleme ne postoji mehanizam usmjeravanja rješenja prema nekom očitom „vrhu“ (problemi odluke između nekoliko opcija).

2.1. Evoluirajuća podloga

Ideja iza evoluirajuće podloge jest jednostavno korištenje mehanizama genetskog algoritma u grafici uz pokušaj zaobilaženja ili iskorištavanja nedostataka samog algoritma. Podloga može biti gotovo bilo što – od čestica, tekstura i ravnina do kompliciranijih modela. Korištenjem nekoliko pravila koja reguliraju kretanje i visinu podloge, ona se može prilagoditi zadatku. Implementacija koristi dvodimenzionalnu plohu koja mijenja oblik kroz vrijeme [13] koristeći najbolju jedinku genetskog algoritma kroz vrijeme [14]. Ista implementacija prikladna je i za generiranje promjenjivih mapa neravnina za objekte (engl. *bump mapping*) u stvarnom vremenu.

2.1.1. Populacija i početno stanje

Svaki genetski algoritam mijenja niz rješenja s ciljem dobivanja nekog boljeg. Populacija veličine jedne jedinke daje potpunu nasumičnost (i nemogućnost primjenjivanja elitizma), a velike populacije dovode do konvergencije u manjem broju vrlo sporih koraka. Idealan broj jedinki u populaciji ovisi o primjeni, s tim da većina primjena zahtijeva barem 20 ili 50 jedinki. Podloga može i ne mora imati eksplicitno definiranu ciljnu funkciju, s time da je u drugom slučaju poželjno je da radi neke prirodne pogreške, pa je eksperimentalno određena populacija u ovom slučaju postavljena na 5 jedinki. Početno stanje podloge je ravnotežno, odnosno sve promjene se primjenjuju direktno na podlogu kakva je bila na početku. Sama ploha u primjeru implementacije je proceduralno generirana ravna površina.

Svaka jedinka predstavljena je poljem koje predstavlja njene vrijednosti za svaku jedinicu površine, što u slučaju tekstura znači polje dimenzija teksture, a u slučaju pomicanja vrhova polje je barem dovoljno veliko da obuhvati sve vrhove. U početnom trenutku sve jedinice su jednake vrijednosti.

2.1.2. Selekcija, križanje i mutacija

Kroz svaku generaciju prati se najbolje ocijenjena jedinka i odabire za križanje. Kod ovako malog broja jedinki dovoljno je uzeti jednu takvu jedinku. Zbog prirode primjene ne postoji velika opasnost od zastajanja u lokalnom optimumu, pa se ta jedinka križa sa svim drugim jedinkama. Ona sama pritom ostaje nepromijenjena, a vrijednosti ostalih jedinki se postave na pola puta prema najboljoj. Ovo je varijanta elitne selekcije u kojoj najbolja jedinka ne samo preživljava nego i u potpunosti određuje slijedeće stanje. Ovo je potrebno zbog toga da prijelaz između generacija izgleda prirodno.

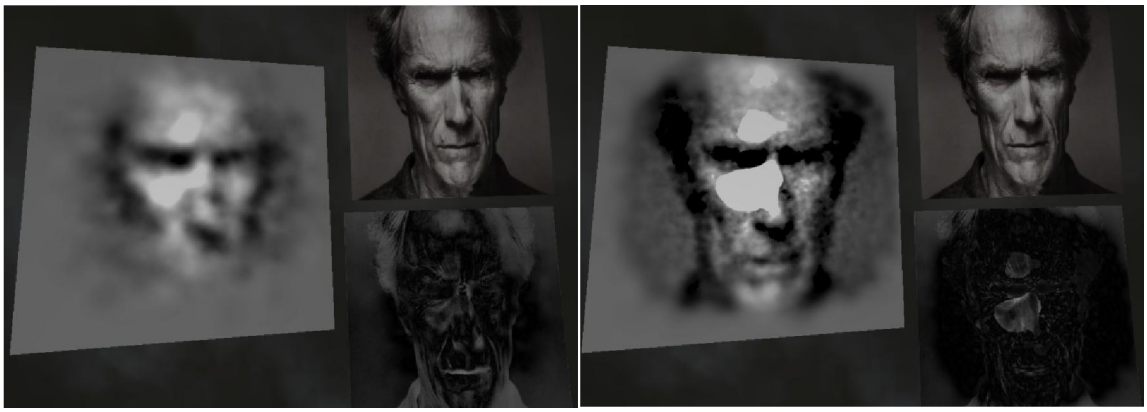
Nakon križanja sve jedinke se mutiraju, što u slučaju ravnih ploha označuje dizanje ili spuštanje nekog područja na plohi (što je reprezentirano kroz povećanje ili smanjenje vrijednosti unutar polja koje predstavlja tu jedinku). Svaka jedinka podvrgava se više mutacija koje uzrokuju izdizanje ili spuštanje površine u krugu s nasumičnim promjerom oko nasumične točke. Vrijednosti bliže središtu tih kružnica mijenjaju se u punom intenzitetu (koji je također nasumično određen), a opadanjem udaljenosti smanjuje se i intenzitet promjene (što je još jedna pojava vidljiva u prirodi). Ovakav mehanizam omogućuje značajne promjene pamćenjem samo položaja središta (koje se može izraziti i kroz samo jedan cijeli broj, jer predstavlja konačnu točku na podlozi) i radijusa kružnice. Bitno je napomenuti da se sve nasumičnosti rade preko Perlinovog šuma i pseudo-nasumičnih generatora brojeva, što znači da je moguće generirati iste rezultate u više pokretanja uz iste parametre, što pokazuje kvalitete različitosti i nepredvidljivosti, a ponovljivosti rješenja.

2.1.3. Evaluacija i prezentacija

Nužna stvar u svakoj iteraciji genetskog algoritma je evaluacija jedinki prema nekoj funkciji ili pravilu koje određuje njihovu dobrotu (engl. *fitness*). Iako ciljna funkcija ne mora biti eksplicitno zadana, ocjenjivanje jedinki je način na koji se usmjeruje djelovanje algoritma. Ciljna funkcija može se i mijenjati kroz vrijeme, a algoritam će se prilagođavati novim zahtjevima. Moguće je definirati razne ciljne funkcije, a za evoluirajuću podlogu osmislio sam jednu eksplicitnu funkciju [15, 16] i jedan skup pravila koja implicitno određuju ponašanje podloge.

Kao provjeru konvergencije algoritma za iscrtavanje na teksturi podlozi je zadan cilj – aproksimacija neke, već postojeće teksture (eksplicitno zadana funkcija). Usprkos sporoj

konvergenciji, algoritam je pokazao da je sposoban kretati se u smjeru rješenja i u konačnici barem približno dostići cilj („kist“ kojim algoritam slika željenu teksturu nije uvijek jednake veličine pa je gotovo nemoguće ispravno naslikati detalje). Koristeći veličinu kista 7% ± 5% veličine teksture, bez ograničenja intenziteta, algoritam je uspio iscrtati donje dvije slike. Na svakoj od slika lijevi pano predstavlja podlogu, gornji desni ciljnu funkciju i donji desni razliku između ciljne funkcije i postignutog rezultata. Bijela površina predstavlja nedostatak ograničenja intenziteta - vrijednost boje podloge u tom području prelazi 1, odnosno 255. Micanje ograničenja domene korisno je za pokazivanje da će usprkos nepogodnom rasponu vrijednosti rezultat i dalje završiti unutar ograničenja originalne domene. Uz elitizam, nakon vrlo velikog broja iteracija, bijela podloga bi čak i u ovom slučaju nestala, a kod ograničavanja vrijednosti algoritam bi konvergirao i znatno brže.



Slika 8 – Aproksimacija eksplicitno zadane funkcije evoluirajućom podlogom (lijevo – nakon 200 iteracija, desno – nakon 65000 iteracija)

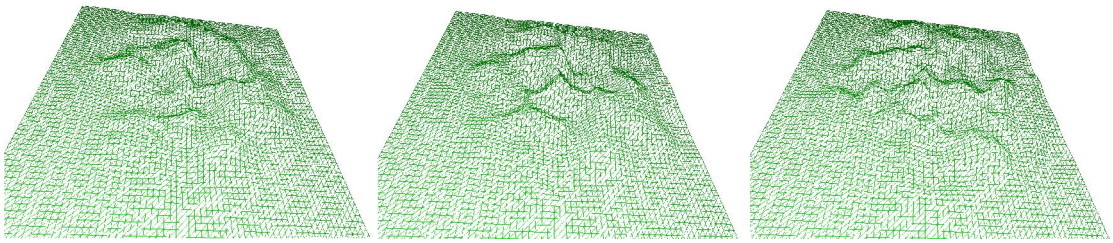
Za slučaj eksplicitno zadane funkcije podloga pokušava dostići željeno rješenje. No, podlozi je moguće zadati i skup pravila koja ona pokušava slijediti. Zbog mutacije funkcija će se konstantno mijenjati, a mali broj jedinki osiguravat će da ta promjena odjednom ne bude prevelika, a i da se funkcija ne vrati prebrzo u neki lokalni optimum.

Primjer tri pravila koja zajedno uzrokuju pravilno ponašanje su:

1. Suma razlika vrijednosti polja i početnih vrijednosti polja nadodaje se kao kazna na ocjenu jedinke
2. Suma razlika vrijednosti polja i vrijednosti polja najbolje jedinke iz prošle generacije nadodaje se kao kazna na ocjenu jedinke

3. Ako je neka vrijednost polja ekstremno visoka ili niska, kazni ju ovisno o tome koliko je različita od prosjeka; dodatno ju kazni za sumu razlike između nje i njenog 4-susjedstva (ili von Neumannovo susjedstvo, odnosno najbliži članovi horizontalno i vertikalno u oba smjera)

Ova pravila demonstriraju potrebu za upravo ovako malom populacijom – uz veću populaciju na nekim primjerima mutacija bi poništavala samu sebe i zbog prvog i drugog pravila stanje podloge bi se stalno vrtilo oko nule, dok je namjena tih pravila očuvanje mase predmeta koji se deformira i prirodna promjena. Ova pravila uz manji broj jedinki postaju tek smjernice – što je manje jedinki to ih algoritam slabije slijedi (s jednom jedinkom rezultati su nasumični i pravila se ignoriraju), tako da uz previše jedinki pravila će se vjerno slijediti.



Slika 9 – Evoluirajuća podloga u nekoliko stanja

Značaj ove podloge je stoga u sposobnosti određivanja razine do koje se pravila koja određuju njeno gibanje slijede – upravo ovo svojstvo zajedno s mutacijom određuje da će se podloga konstantno mijenjati, s time da će u svakom trenutku nastojati zadovoljiti sva zadana pravila. Moguće je napraviti skupove pravila koje je potrebno slijediti više ili manje i time stvarati nove načine promjene podloge.

Za daljnji razvoj i iskorištavanje punog potencijala ove podloge bilo bi zanimljivo prenijeti veći dio posla na grafičku karticu ili barem parelelno obrađivati jedinke. Moguće su i brojne druge optimizacije poput računanja pogreške samo na promijenjenim dijelovima. Uvođenje varijanti podloge je također vrlo jednostavno, poput uvođenja podskupova jedinki koje obrađuju pojedina pravila te križanja između više skupova jedinki za dobivanje one koja se prikazuje. Donja tablica prikazuje u kojem rasponu trajanja izvođenja se nalazi otprilike 95% jedinki nekog tipa. Rezultati su dobiveni na osnovi 1000 uzoraka.

Zadana funkcija	*Eksplicitni slučaj (tekstura)*		*Implicitni slučaj (ravnina)*
Veličina polja	512x512	64x64	64x64
Evaluacija	44 – 47 ms	0.6 – 0.8 ms	1.6 - 1.8 ms
Prikaz	118 - 120 ms	2.7 – 2.8 ms	1.8 – 2.0 ms
Križanje	40 – 54 ms	0.8 – 0.9 ms	0.8 – 0.9 ms
Mutacija	8 – 80 ms	0.8 – 2.0 ms	0.7 – 2.2 ms
Ukupno iteracija	230 – 290 ms	5.5 – 6.4 ms	5.1 – 6.7 ms

Tablica 1 - Brzina iteracija algoritma ovisno o ciljnoj funkciji i tipu podloge

2.2. Evolucija 3D objekata

Kroz ovaj primjer demonstrirao bih jedan drugačiji pristup oblikovanju grafičkih objekata koji koristi genetski algoritam. Korištenje genetskih algoritama je često neučinkovito kod složenijih primjera jer se loše skaliraju s porastom složenosti – mutacija je nasumična i za složenije primjere često presporo (ako uopće) usmjerava evoluciju u pravom smjeru.

Kako bi bilo moguće korištenje genetskih algoritama potrebna je ciljna funkcija (koja, iako ne treba biti eksplicitno definirana, mora biti nešto u što sama evolucija teži). Za ovaj primjer automatizirati ću proces određivanja sudarača (engl. *collider*) kao aproksimaciju složenih trodimenzionalnih modela. Detekcija sudara u grafici je vremenski skup proces čija brzina ovisi o složenosti sudarajućih modela. Zbog toga se često koriste složeni modeli za iscrtavanje a pojednostavljeni za sudaranje. Cilj pojednostavljenih modela je da što bolje opisuju složeni model koji promatrač vidi, tako da je opisivanje složenog modela jednostavnima zapravo funkcija kojoj genetski algoritam u ovom slučaju pokušava naći optimum. Jednostavni sudarači najčešće su stvoreni od više sfera (detekcija s kuglom može se jednostavno detektirati), liste kocaka ili kvadara (jer su vrlo jednostavni) ili smanjivanjem složenosti direktno iz originalnog, složenog modela.

2.2.1. Populacija i početno stanje

Svaka jedinka populacije je zasebni sudarač sastavljen od proizvoljnog broja kugli ili nekih drugih jednostavnih objekata. Za svaku kuglu pamte se 4 podatka – položaj (x, y, z) i polumjer. Za druge objekte, poput elipsi, potrebno je pratiti 9 podataka – položaj (3), rotaciju (3) i skaliranje (3). Modifikacijom tih varijabli mijenja se oblik sudarača i kroz genetski algoritam opisuje ciljani model.

Za razliku od prošle primjene genetskih algoritama, u ovom se slučaju početno stanje namješta na poželjne vrijednosti. Koristi se algoritam k srednjih vrijednosti (engl. *k-means algorithm*). Broj centara algoritma jednak je broju jednostavnih modela koji čine jedan sudarač. Uzorci koji se grupiraju su vrhovi složenog modela. Konačni rezultati (dobiveni centri) postati će središta kugli (ili nekih drugih objekata kojima se aproksimira složeni) za svaku jedinku genetskog algoritma.

Za vrhove $x^{(1)}, \ldots x^{(m)}$ algoritam pronalazi k centroida i oznaku pripadnosti c za svaki vrh:

Inicijaliziraj centroide $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$ nasumično

Ponavljaj do konvergencije

Za svaki i

$$c^{(i)} = arg\min_j \left\|x^{(i)} - \mu_j\right\|^2$$

Za svaki j

$$\mu_j = \frac{\sum_{i=1}^{m} 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{c^{(i)} = j\}}$$

Uvjet zaustavljanja algoritma određen je na dva načine: on staje ukoliko u nekoj iteraciji nije došlo do nikakvih pomaka centroida ili ukoliko je istekao zadani maksimalni broj iteracija (kod složenijih modela, dok se traži više centroida može doći do beskonačnih petlji, no pošto u ovom koraku ne tražimo savršeno rješenje i već nekoliko iteracija ovog algoritma poslužiti će kao dovoljno dobra početna točka za genetski algoritam).

2.2.2. Selekcija, križanje i mutacija

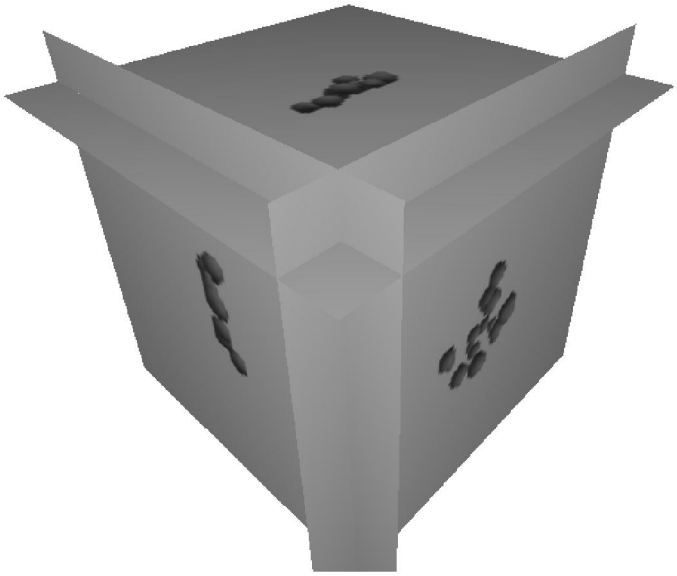
I u ovoj inačici genetskog algoritma koristim elitizam kako bi se očuvala najbolja jedinka, ali se ne križaju sve jedinke (pojedina jedinka ima samo vjerojatnost križanja s najboljom) niti se sve mutiraju (svaki dio svake jedinke ima vjerojatnost mutacije).

Pri križanju novostvorena jedinka koristiti će prosjek vektora iz kojih je stvorena, s time da (za slučaj kugli, na primjer) položaj i polumjer imaju zasebne vjerojatnosti za križanje pa je moguće da se promijeni samo jedan od tih podataka. Također, pošto je poznato koja kugla je zadužena za pokrivanje kojeg područja (zbog korištenja algoritma k srednjih vrijednosti), križaju se samo neke kugle s kuglama koje su počele na istom položaju, jer bi one trebale predstavljati isti gen.

Za svaku varijablu kugle se odvojeno provjerava vjerojatnost mutacije, a ona uključuje pomak jedinke u prostoru i promjenu promjera kugli (za složenije objekte mijenja se i rotacija i veličina zasebno za sve tri dimenzije).

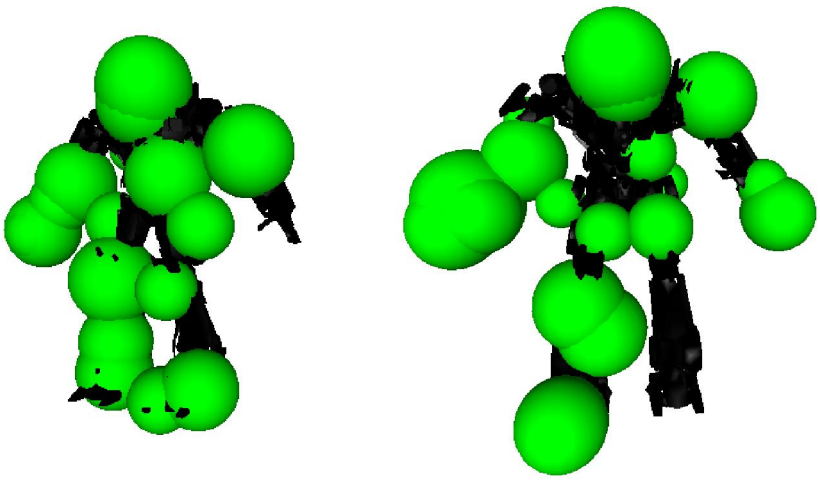
2.2.3. Dohvat podataka i evaluacija

Podaci koji se dohvaćaju su zapravo projekcija složenog objekta na tri teksture – po jedna za svaku os (x, y, z). Dvije su varijante implementacije – jedna u kojoj se dohvaćaju projekcije objekta i svakog sudarača zasebno i druga u kojoj se dohvaća samo projekcija složenog objekta jednom. Prednost prve implementacije je u tome što je za sudarače jednostavno moguće postaviti bilo koje geometrijsko tijelo (čak i kompliciranije modele) jer se dohvaća njihova projekcija i gleda preklapanje s projekcijom složenog objekta. Prednost druge implementacije je u tome što je nešto brža jer se uz poznate položaje, rotacije i skalu tih dijelova može izračunati preklapanje s objektom koji se opisuje bez potrebe za stvaranjem teksture tih objekata. Takva implementacija je, s druge strane, puno kompliciranija za izvesti kad se koristi rotacija i skaliranje po svakoj osi (osobito za proizvoljne objekte), pa je ovdje zbog jednostavnosti implementirana samo za kugle (koje se jednoliko skaliraju po svim osima i invarijantne su na rotaciju) – ideja algoritma ostaje ista.



Slika 10 - Projekcije objekta na tri plohe usred evolucije

U oba slučaja jedinka se kažnjava za prekrivanje prostora koji se ne nalazi unutar objekta i za neprekrivanje prostora koji se nalazi unutar objekta. Dodatno, za središte svakog drugog jednostavnog tijela koje se nalazi unutar granica bilo kojeg takvog tijela jedinka se dodatno kažnjava. Evaluacija se ne provodi kroz algoritam k srednjih vrijednosti, pa će početne vrijednosti biti prilično nasumične.



Slika 11 – Sudarač (zeleno) nakon 100 iteracija u dva različita izvođenja

Na gornjoj slici prikazan je rezultat nakon 100 iteracija. U evoluciji sudjeluje 5 jedinki, a svaka se sastoji od 20 kugli. Objekt koji se pokušava opisati je humanoidni robot. Za iskoristiv rezultat potrebno je više desetaka tisuća iteracija, što je vremenski zahtjevno.

Za razliku od prošle primjene, stvaranje *collidera* mora konvergirati u neku konačnu točku – postoji optimum funkcije. Smanjenje ukupne greške u ovom slučaju, nakon inicijalizacije algoritmom k-means, se događa vrlo sporo. Ponekad je potrebno i nekoliko tisuća generacija da se pronađe bolja jedinka, a prednost koja se stekne k-means inicijalizacijom u odnosu na nasumično postavljana tijela ubrzava algoritam za više tisuća iteracija (što može potrajati i do nekoliko sati). U donjoj tablici prikazano je vrijeme izračuna iteracija za obje varijante implementacije, jednu koja se oslanja na grafički prikaz objekata i drugu koja to ne čini. Uz bolju optimizaciju, varijanta bez grafičkog prikaza bila bi znatno brža.

	Dohvat	*Evaluacija*	*Križanje*	*Mutacija*	*Iteracija*
S grafičkim prikazom	70 ms	25 ms	0.8 ms	4.5 ms	~100 ms
Bez grafičkog prikaza	65.8 ms	22 ms	0.8 ms	4.4 ms	~95 ms

Tablica 2 – Trajanje pojedinih dijelova iteracije kod trodimenzionalne evolucije

2.3. Usporedba evolucijskih i klasičnih algoritama

Za ovu svrhu evolucijski algoritmi su i dalje iznimno spori. Čak i uz prilično dobre mogućnosti paraleliziranja 3D oblikovanja objekata, klasični algoritmi postižu impresivnije rezultate u manje vremena. Još jedna negativna strana evolucijskih algoritama je u tome što ne konvergiraju deterministički – u bilo kojem konačnom broju iteracija (pod pretpostavkom da algoritam nije zastao u lokalnom optimumu) greška će gotovo uvijek biti različita, kao i jedinke populacije, pa je moguće da dva pokretanja istog algoritma daju jednako dobar rezultat u razmaku od nekoliko sati (ova razlika raste s brojem dimenzija, tako da se za evolucijsku podlogu i neće osjetiti dok je za trodimenzionalni primjer vidljiva). Kao deterministički odgovor na ovaj problem opisao bih jedan zanimljiv klasični algoritam koji, iako ne postiže identičan cilj kao i ovaj evolucijski, je znatno primjenjiviji i u novije vrijeme se često koristi u proceduralno generiranim svjetovima. Algoritam je zasnovan na pojmu voksela (engl. *voxel* = *volume* + *pixel*), koji su inače zanimljivi i za korištenje u genetskim algoritmima jer smanjuju veličinu prostora te olakšavaju evaluaciju ispunjavanjem tijela. Jedna od njihovih najvećih mana je u tome što su relativno spori u odnosu na konvencionalne metode i algoritme prikaza svijeta. Vjerojatno najpoznatiji algoritam vezan uz ovo područje je algoritam *marching cubes,* no implementacija ovdje

koristi jedan jednostavniji i intuitivniji algoritam koji je dovoljan za pokazivanje ideje voksela i usporedbe takvih algoritama s evolucijskim.

2.3.1. Vokselizacija i vokseli

Ideja iza voksela je volumni prikaz objekta. Pošto je svaki individualni voksel zasebni objekt koji graniči s drugim identičnim vokselima, oni se mogu koristiti za razne simulacije fizike, no u zadnje vrijeme su vrlo popularni u računalnim igrama. Proces prikazivanja objekta kroz voksele zove se vokselizacija (engl. *voxelization, mesh voxelization*). Vrijednost voksela zapisuje se u trodimenzionalna polja koja predstavljaju ispunjenost prostora, pa je najpraktičnije tijelo za njihov prikaz kocka jer potpuno ispunjava svijet postavljanjem na pravilnu trodimenzionalnu mrežu. Svaki voksel može biti opisan i vlastitom bojom i brojnim drugim oznakama.

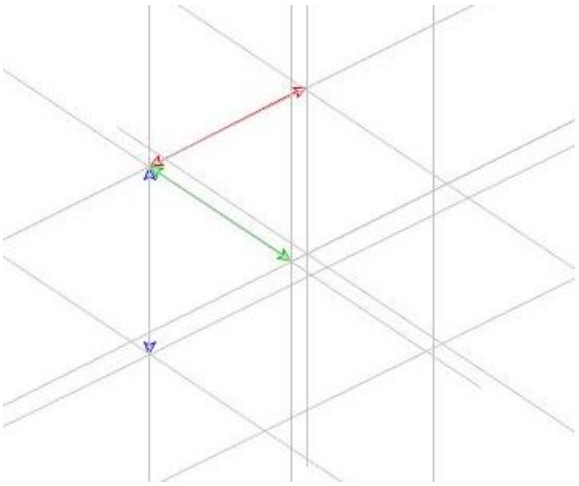
Slika 12 – Igra koja koristi voksele

Postoji više metoda vokselizacije, od direktnog modeliranja preko ulaznih uređaja, preko projekcije objekta na pomičnu podlogu koja se kreće kroz sam objekt, do metoda praćenja zrake (engl. *ray tracing*) i raznih drugih [17, 18, 19, 20]. Metoda koja je ovdje korištena je jedna od sporijih – ona koja koristi praćenje zrake [17] – ali istovremeno i jedna od onih koje je najlakše modificirati da stvara drugačije voksele.

Kao inicijalizacija algoritma, prostor se prvo podijeli na dijelove – eksperimentalno je vrlo dobro za dimenziju jednog piksela uzeti dvadesetinu prosjeka dimenzija složenog objekta, ali se veličina voksela jednako dobro može definirati i ručno. Veličina voksela ovisi o

brzini izvođenja algoritma i kasnijoj brzini iscrtavanja voksela, tako da veliki vokseli mogu poslužiti kao dobra aproksimacija nekog složenog objekta za potrebe sudara.

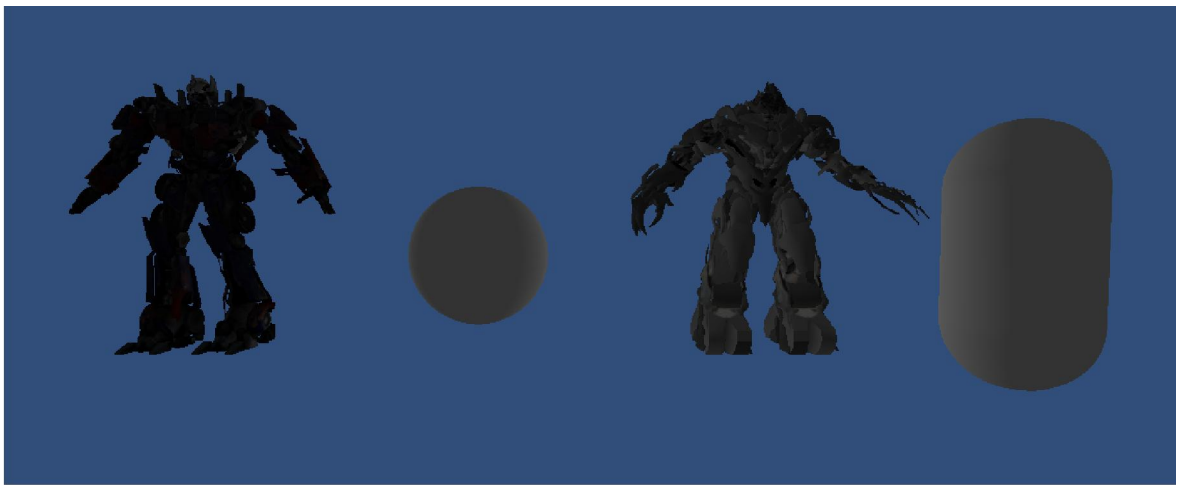
Metoda koja je korištena prati zraku uzduž svih dimenzija mreže koja se stvara unutar granica objekta. Ukoliko zraka siječe objekt, na tom i okolnim položajima se registriraju vokseli. Za svaku koordinatu mreže za kvalitetnu vokselizaciju potrebno je izvesti praćenje zrake šest puta – jednom u smjeru svake osi i natrag, s tim da se zraka za neku koordinatu prati tek do duljine dimenzija jednog voksela. Ukoliko neka zraka prolazi kroz objekt, sve koordinate oko nje označuju se kao ispunjene i više ih nije potrebno provjeravati. Naravno, uz bolju granularnost ovaj postupak davati će bolju aproksimaciju objekta koji se vokselizira. Nakon što je proces obavljen za sve točke trodimenzionalne mreže, na ispunjenim mjestima iscrtavaju se vokseli. Ispunjenost polja može se jednostavno i brzo pratiti kroz *byte*, *bool* ili *BitArray* tipove podataka.



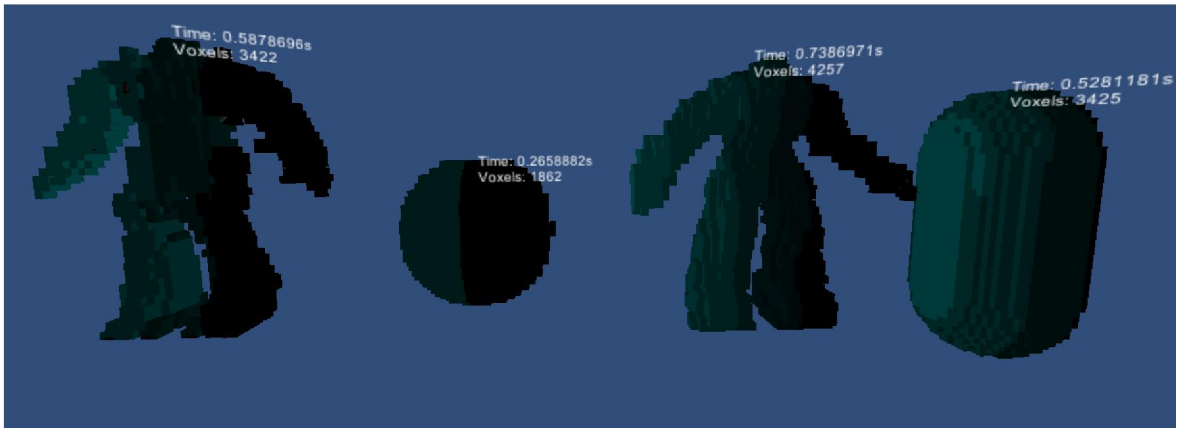
Slika 13 – Praćenje zrake kroz mrežu

Jedna od glavnih razlika između izrade modela od voksela i poligonskih modela je u tome što vokseli ispunjavaju i unutrašnjost samog modela. Iako za potrebe sudaranja ovo nije nužno, algoritam za ispunjavanje unutrašnjosti modela je također jednostavan – naime, osnovni algoritam postavlja voksele samo na granice objekta, pa je potreban drugi prolaz za definiranje unutrašnjosti (kod nekih algoritama ovo se izvodi u istom prolazu). Kod drugog prolaza algoritam prolazi kroz prostor kao i prije, ali nakon što uđe u objekt (prođe prve popunjene voksele) pošalje zraku u smjeru u kojemu se kreće (i natrag) kako bi provjerio u kojem je smjeru okrenut slijedeći poligon – ako takvog poligona nema, preskoči tu liniju; ako je poligon okrenut u smjeru trenutnog položaja algoritma, prebaci se na tu lokaciju. Ako je poligon okrenut od trenutnog položaja, ispuni a polja do tog

poligona. Alternativno, moguće je pamtiti u kojem smjeru je okrenut poligon nekog stupca ili retka kao dodatni podatak zapamćen pri stvaranju ljuske.



Slika 14 – Modeli za vokselizaciju



Slika 15 – Vokselizirani modeli s trajanjem vokselizacije površine i brojem voksela

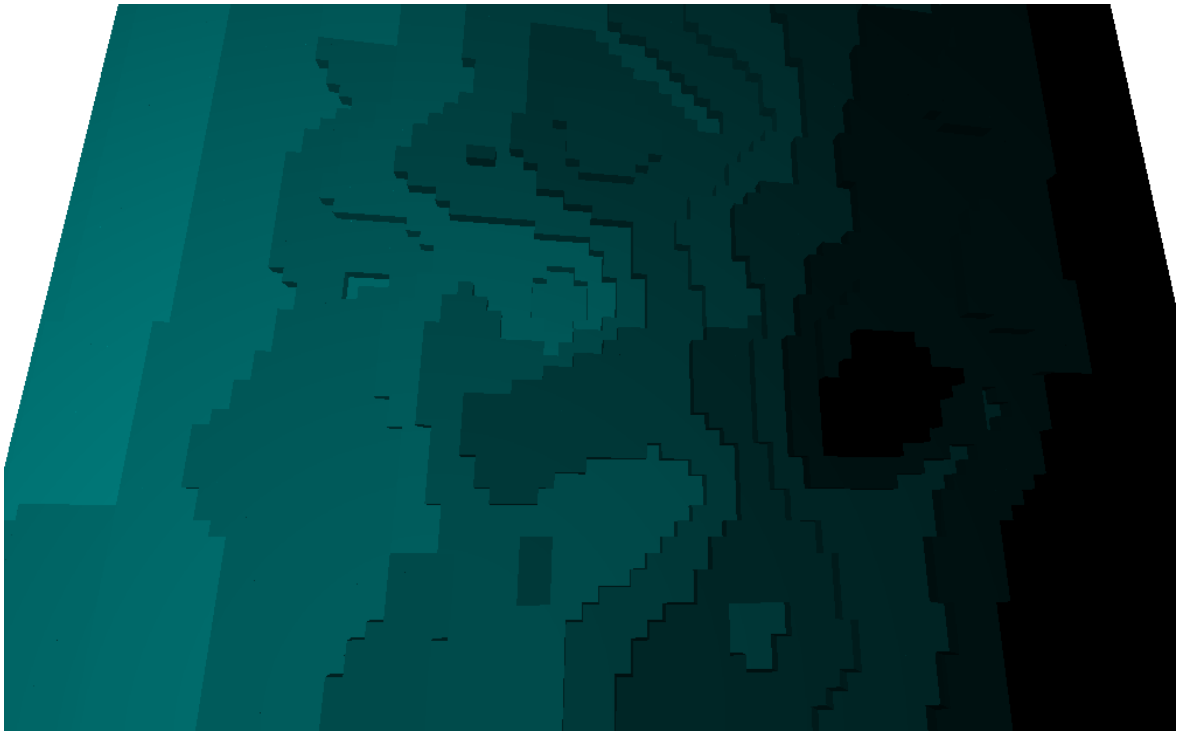
Usprkos iznimno velikom broju objekata, svjetovi napravljeni od voksela mogu se pokretati u stvarnom vremenu zahvaljujući brojnim optimizacijama. Jedna od osnovnih takvih optimizacija uključuje spajanje više voksela u jedan spremnik. Taj spremnik se nakon toga brine o individualnom adresiranju voksela, a iscrtava sve voksele jednim pozivom grafičke kartice što značajno ubrzava prikaz. Za veće scene postoji čitava hijerarhija spremnika, a za svjetove u kojima je potrebno mijenjati voksele (najčešće za animaciju objekata) spremnici su manji i organiziraju se dinamički. Nedostupni ili nevidljivi vokseli (što uključuje i pregradne strane susjednih kocaka) su najčešće uklonjeni prije slanja poziva grafičkoj kartici, a u slučaju proceduralno generiranog svijeta udaljene jedinice mogu se računati i u trenutku pristupa.

Čak i bez brojnih optimizacija, uz korištenje punih tijela i uz iscrtavanje svih nevidljivih poligona, vokselizacija nekog objekta do gotovo proizvoljne preciznosti traje znatno kraće od već nekoliko generacija genetskog algoritma koji oblikuje pojednostavljeni sudarač za neki objekt iz više jednostavnih objekata. Na slijedećoj tablici pokazano je trajanje vokselizacije za isti objekt, ovisno o veličini voksela. Povećanjem broja voksela povećava se i broj njihovih spremnika tako da vrijeme raste neznatno brže od linearnog u odnosu na broj voksela (za navedene primjere to je 0.17 – 0.25 ms po vokselu).

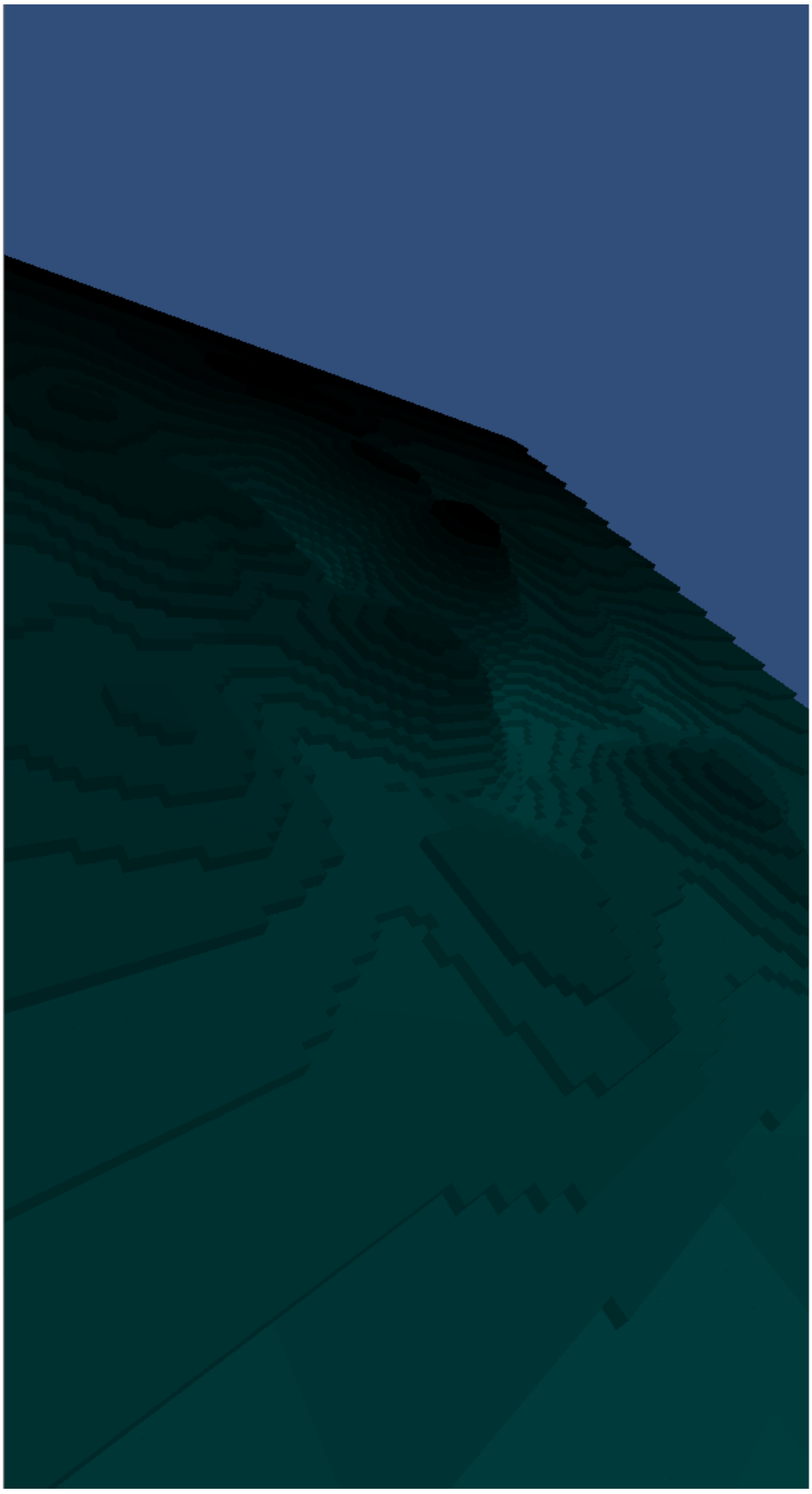
Veličina voksela	*Broj voksela*	*Trajanje vokselizacije (s)*
0.1	663	0.1149
0.05	3452	0.5988
0.02	29120	6.5113
0.015	55361	13.9851

Tablica 3 – Ovisnost broja voksela o njihovoj veličini i trajanja vokselizacije o broju

Na slijedećim slikama prikazana je kombinacija rezultata genetskog algoritma iz poglavlja evoluirajuće podloge i vokselizacije.



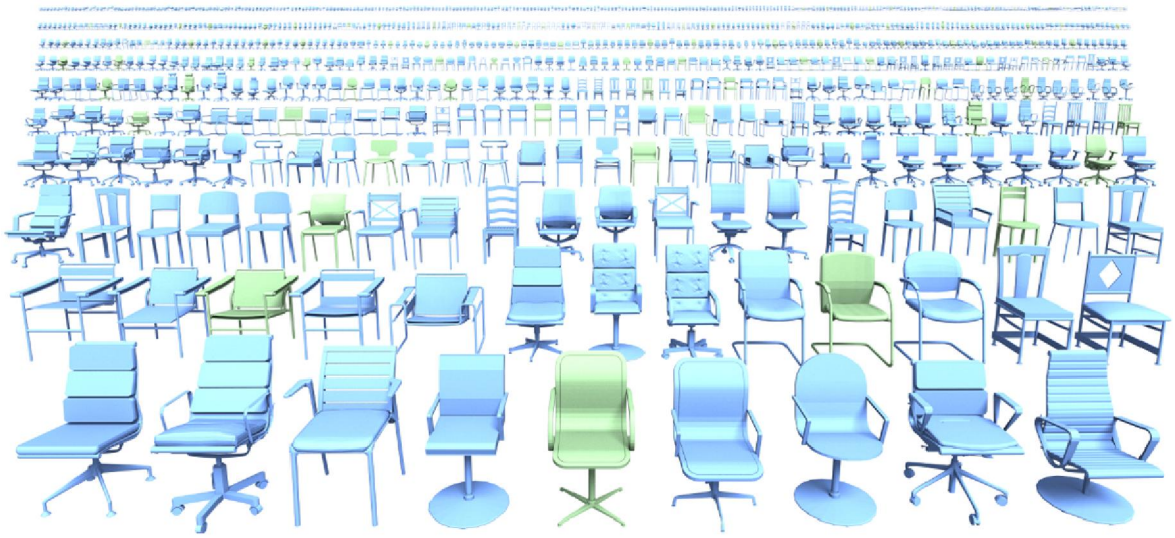
Slika 16- Vokselizirana evoluirajuća podloga odozgo



Slika 17 – Vokselizirana evoluirajuća podloga

3. Sinteza modela

Sinteza modela uključuje spajanje više gotovih dijelova modela u smislenu cjelinu, s ciljem stvaranja velike količine originalnog sadržaja iz male količine podataka. Objekti za koje se koristi ovakva metoda su obično generirani masovno – ili odjednom (prijevozna sredstva, gradovi, ljudi, karte i teren) ili individualno i periodično (oružje i oruđe, namještaj u nekoj sceni). Ovo je prilično široka definicija i obuhvaća veliki dio proceduralno generiranog sadržaja, pa se ovako općenit pojam i ne koristi vrlo često.



Slika 18 – Plavi objekti su sintetizirani iz dijelova zelenih [21]

Ne postoji propisani način spajanja modela i različite implementacije će vjerojatno funkcionirati na različite načine, ali sve će na neki način odraditi slijedeći skup zadataka:

1. Predobrada objekata – stvaranje i semantičko odvajanje modela u skupine
2. Obrada objekata – dijeljenje modela na najmanje razlikovne dijelove, skaliranje
3. Sinteza – spajanje objekata, nasumično ili prema nekoj funkciji dobrote
4. Optimizacija – detaljno namještanje odnosa objekata, micanje nepotrebnih poligona

Prvi korak najčešće se vrši ručno i s gledišta proceduralnog generiranja grafičkih objekata nije bitan za cijeli proces, osim ako se ti modeli generiraju proceduralno u kojem slučaju se sastoje od nekih vrlo jednostavnih dijelova. Drugi korak se i dalje izvršava ručno (obično pri stvaranju objekata), iako postoje neki algoritmi (kao ranije korišteni algoritam k-means) koji bi mogli odraditi isti posao s velikom preciznošću. Ovaj posao može sadržavati i

generiranje stabla kroz koje se model dijeli na razne dijelove. Kroz čvorove stabla zapisan je odnos dijelova modela, najčešće na slijedeći način: korijen stabla predstavlja cijeli model, a njegova prva djeca predstavljaju prvu (i obično jedinu) razinu detalja – ukoliko se i ti dijelovi mogu podijeliti, svaki dalje ima svoju djecu koja predstavljaju njegove dijelove. Kod sklapanja objekata odabire se razina detalja na kojoj se objekti spajaju, pa se i neki dijelovi mogu napraviti sintezom (svaki čvor definira i semantiku nekog dijela, tako da se ne spajaju nespojive komponente). Različiti modeli mogu se sastojati od različitog broja istog tipa dijelova, u kojem slučaju se svi ti dijelovi računaju kao jedna semantička jedinica (na primjer, stol može imati više ili manje od 4 noge, svijećnjak gotovo proizvoljan broj svijeća, ali neki električni uređaj može i ne mora imati žicu za struju).

Treći korak predstavlja srž procesa jer se kroz njega odabiru dijelovi koji će se koristiti u konačnom modelu. Iako je ideja proceduralnog generiranja objekata da ovaj proces bude automatiziran, postoji i varijanta sinteze modela u kojoj sam korisnik odabire dijelove koje želi koristiti (to je jedan od načina stvaranja *user mediated contenta*). Ovaj korak se također sastoji od najvećeg broja varijanti, od kojih ću nabrojati nekoliko. Modeli izgrađeni nasumičnim biranjem odgovarajućih komponenti i oni izrađeni posredovanjem korisnika su jednostavniji, a izgradnja probabilističkog modela [21] poklapanja raznih dijelova je složeniji način konstrukcije rezultata. Vrlo zanimljiv način generiranja sintetiziranih objekata koji neznatno odudara od ovih je čisto proceduralno generiranje dijelova preko funkcija, kao što je napomenuto u prvom poglavlju. Ovakav način generiranja koristi se kod objekata složenih od jednostavnih djelova koji se često ponavljaju ili jednostavno miješaju. Primjer za nasumično biranje komponenti bio bi stol čija se duljina određuje, ali dizajn (broj nogu, reljef) se mijenja ovisno o toj duljini. Takvi modeli mogu koristiti i razne gramatike, svaka od kojih može posjedovati različita pravila za spajanje. Posredovanje korisnika može se naći u mnogim igrama u kojima se slažu razni oblici – od slaganja letjelica do čitavih živih bića od dijelova (igra *Spore*). Probabilistički model, vjerojatno najbolje opisan kroz [21], pokušava odabrati komponente učenjem povezanosti između geometrijskih i semantičkih karakteristika dijelova [22, 23]. Primjer za slaganje proceduralnih komponenti bio bi slaganje gotovih likova iz lego kocaka, gdje bi se oblik gotovih likova mogao određivati i nekim drugim (na primjer evolucijskim) algoritmom.

Konačni korak, optimizacija, je također značajan dio sinteze za koje se koriste algoritmi koji optimiziraju međusobni odnos dijelova kako bi poklapanje izgledalo prirodno. U

gotovo svim slučajevima sadržaja koji generira korisnik (i mnogim drugima), određivanje točnog odnosa dijelova je napravljeno automatski – kad se komponenta postavi u blizinu druge, njena točka spoja postavlja se u točku spoja te druge komponente. Ukoliko o samim dijelovima nije dano toliko podataka, optimizacija ih pokušava spojiti s minimalnom kolizijom.

3.1. Izgradnja automatskim odabirom komponenti

Ukoliko se komponente nekog modela razlučuju upravo do detalja na kojima nema nikakvih semantičkih razlika u sastavu modela, moguće je jednostavnom zamjenom dijelova između modela napraviti sustav koji učinkovito stvara nove modele. Radeći s besplatnim modelima koji nisu jednoliko napravljeni, obrada početnih modela uzima puno vremena, a daje rezultate koji se još znatno mogu unaprijediti.

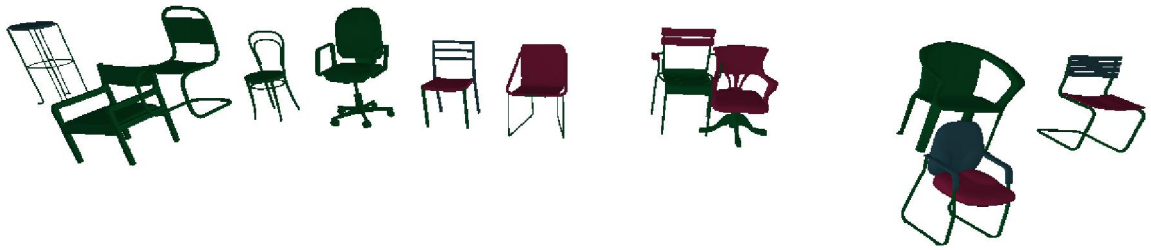
Slika 19 – Početni modeli ograničene veličine na gornju granicu

Neki od problema su:

- Nedovoljna ili pretjerana granuliranost – neki modeli su razrađeni u najmanje detalje, a neki se sastoje od samo jednog dijela (takve modele je prvo potrebno ručno podijeliti na dijelove)
- Skaliranje – svi modeli su različito skalirani – ovaj problem se donekle rješava skaliranjem svih predmeta na istu vrijednost prosjeka njihovih dimenzija
- Centriranje – modeli su različito centrirani, što stvara problem ukoliko se dijelovi spajaju ovisno o lokalnim koordinatama modela
- Rotacija – još jedan problem koji je potrebno riješiti ručno zbog toga što nisu svi modeli orijentirani u istom smjeru – i to ponekad po nekoliko osi

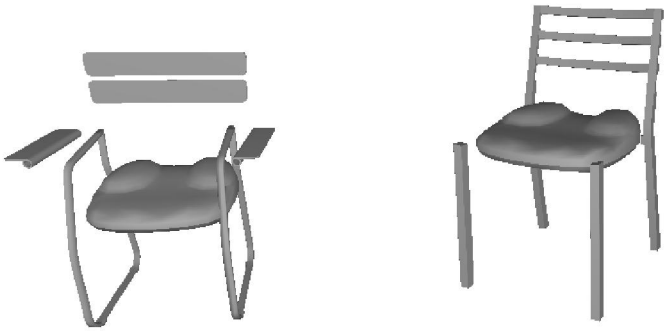
Od 12 početnih modela pola ih je bilo pretjerano granulirano, 5 napravljeno od samo jednog dijela i samo jedan model bio je približno dobre granuliranosti.

Ukoliko se model dijeli na tri dijela koji se u jednom rješenju pojavljuju najviše jednom, dvanaest modela može stvoriti preko 1500 dijelova (ovisno o tome može li se neki dio izostaviti i sadrži li više modela iste dijelove).



Slika 20 – Prvi rezultati obrade bez centriranja predmeta

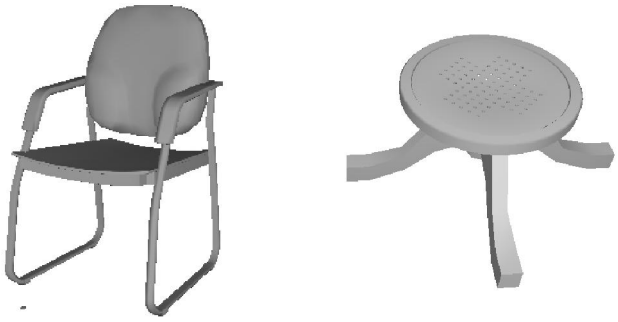
Rezultati su prikazani na slijedećim slikama. Oni su dobiveni bez optimiziranja skaliranjem dijelova i preklapanjem spojnih točaka, no uz manje modifikacije moguće je dobiti iskoristiv model. Konačni rezultat uvelike ovisi o početnoj obradi podataka – uz jednoliko centriranje modela i označavanje dijelova moguće je dobiti znatno bolji rezultat koji zahtijeva tek minimalnu optimizaciju. Ipak, za savršeno skladne rezultate bilo bi se potrebno vratiti na razinu ručnog modeliranja početnih modela kako bi stolci mogli biti podijeljeni na proizvoljnu razinu detalja.



Slika 21 – Neusklađeni modeli stolaca dobiveni spajanjem

Ti modeli prikazuju i neke od većih problema optimizacije. Slika nalijevo prikazuje lošu podjelu jednog od početnih stolaca na dijelove jer naslon nije pravilno povezan s ostalim dijelovima, te loše skaliranje tih dijelova. Slika nadesno prikazuje loše slaganje između dva modela – uz skaliranje i spajanje modela u nekim ključnim točkama ovo je moguće ispraviti. Međutim, ispravljanje svih ovih grešaka istovremeno (i na neki način izrada algoritma univerzalnog spajanja modela) je kompliciraniji proces koji je jednostavnije

izostaviti, pa kasnije ručno ili automatski namještati dijelove. Na donjim slikama prikazana su uspješnija spajanja modela. Lijevi model sastavljen je od tri dijela, od kojega su dva uzeta s istog početnog stolca, a desni od dva dijela s različitih modela.



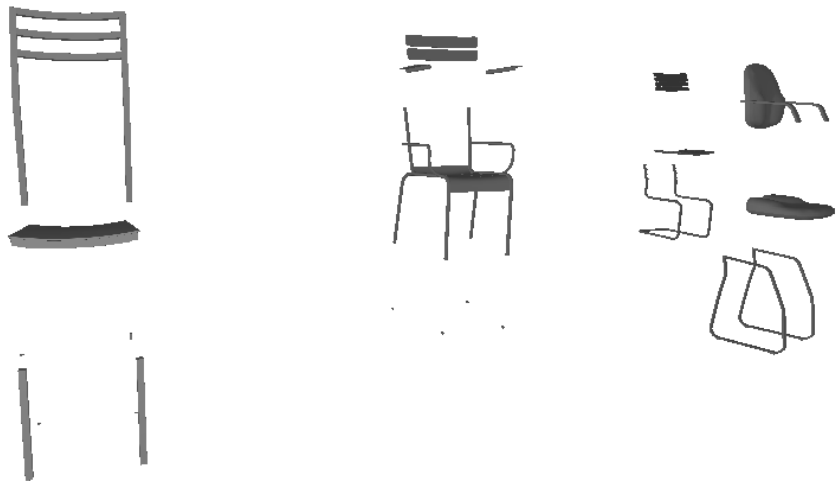
Slika 22 – Modeli dobiveni uspješnim spajanjem

Ovakvi rezultati se u implementaciji izvedenoj u ovom radu dobivaju na slijedeći način. Nakon osnovne obrade modela označene su razine podjele na dijelove za svaki model (0 za samo jedan dio, 1 za svaki dodatni dio na koji je model podijeljen), pri čemu je potrebno naglasiti da su svi modeli u ovom slučaju registrirani kao da imaju samo jednu razinu detalja. Odabran je nasumični model i njegova razina detalja. Taj model služi kao nacrt za konačni model, iako niti jedan njegov dio ne mora biti korišten. Potrebni dijelovi traže se po modelima s istom podjelom na dijelove. Neki dijelovi mogu se koristiti u više podjela – na primjer, jedan stolac podijeljen je na gornji dio i noge a svi ostali razlikuju tri dijela (noge, sjedište i naslon). Ukoliko je kao osnova odabran upravo on, biti će uzet njegov gornji dio, a noge mogu biti njegove ili noge bilo kojeg stolca koji ima označen dio nogu. Ukoliko je odabran neki model koji razlikuje tri dijela, za komponentu nogu može biti odabran i dio koji se nalazi na dvodijelnom modelu.

Stvaranje jednog takvog modela traje u prosjeku 48 ms, s vrlo malom varijacijom za broj dijelova. Ipak, prema [21], usprkos drugačijem načinu sinteze modela, vidljivo je da se rastom broja modela ovaj proces višestruko usporava i da je sinteza svih mogućih kombinacija dijelova za već nekoliko desetaka modela i dijelova neizvediva u stvarnom vremenu. Usprkos tome, ovaj način stvaranja objekata postaje sve popularniji jer su zahtjevi igara obično znatno manji od istovremenog generiranja svih kombinacija velikog broja modela, a ovakve metode znatno doprinose osjećaju jedinstvenosti i micanju osjećaja generičnosti koji dolazi s gotovo svim proceduralno generiranim sadržajem.

3.2. *User mediated content* (UMC)

Iako je sadržaj dobiven posredovanjem korisnika širok pojam, jedan njegov veliki dio zasniva se upravo na sintezi modela. Naime, brojne igre koriste sintezu modela kako bi igrač spajao razne likove, od osnovnih geometrijskih oblika do kompliciranih organizama (*Spore, Impossible Creatures*). Vidljivi rezultati ovakvog načina rada identični su automatskom spajanju modela, a razlika je u tome što umjesto nasumičnog odabira računala korisnik ima pregled nad opcijama za sve dijelove pa sam slaže model po nekim nacrtima. Neke od najvećih prednosti ovakvog načina rada su jednostavnost, učinkovitost (odabir i fino podešavanje namještava korisnik što značajno štedi vrijeme i prostor) te, zbog velikog broja opcija, dobivanje vrijednosti u ponovnom prolaženju igre (engl. *replay value*). Razlog zašto se ovakav način proceduralnog generiranja sadržaja ne koristi češće je u ograničenju složenosti zahtjeva postavljenih direktno korisniku i tome što se ne može uvijek primjeniti (ideja nekih igara onemogućuje korisniku pristup nekim modelima i slično).



Slika 23 – Upravljanje dijelovima stolaca (korisnik)

Samo područje stvaranja grafičkih objekata posredovanjem korisnika uključuje i mijenjanje parametara svih dijelova, čime korisnik direktno uređuje neki objekt prema svojim željama. Način uređivanja sadržaja na ovaj način koristi se ponajviše u računalnim igrama uloga (engl. *role-playing games*) u kojima je izgled objekta moguće mijenjati u ovisnosti o brojnim parametrima – u nekim novim igrama kroz stvaranje avatara moguće je mijenjati stotine različitih parametara što rezultira velikom razlikom u izgledu, pa je iznimno popularno u igrama s više igrača.

Zaključak

Područje proceduralnog generiranja sadržaja široko je i obuhvaća brojne metode koje se danas koriste. U ovom radu opisan je dio tog područja koji se odnosi na grafičke objekte, uz osvrt na primjenu genetskih algoritama umjesto klasičnih.

Proceduralno generiranje objekata brz je način stvaranja sadržaja koji rasterećuje memoriju tvrdog diska pod cijenu procesorskog vremena. Algoritmi stvaranja proceduralnog sadržaja osobito su korisni kod sličnih objekata koji se ponavljaju više puta – različitost između takvih objekata opisuje se kroz kod umjesto veliki broj pohranjenih objekata.

Evolucijski algoritmi u proceduralnom generiranju objekata modeliraju nasumičnost koja je potrebna u većini metoda. Korištenjem genetskog algoritma i optimizacijom ciljne funkcije moguće je kroz pravila usmjeriti evoluciju konačnog modela u željenom smjeru.

Klasični algoritmi često su precizniji i znatno se brže izvode, no za neke primjene moguće je koristiti evolucijske algoritme koji mogu biti jednostavniji, intuitivniji i općenitiji od klasičnih.

Općenito, proceduralno generiranje grafičkih objekata grana je računalne grafike koja se tek razvija, ali već se koristi u gotovo svim većim projektima. Zbog toga očekujem značajan razvoj alata i metoda u ovom području u bliskoj budućnosti, što će dodatno popularizirati ovaj, već zanimljiv i koristan način stvaranja grafičkih objekata. Unošenje evolucijskih algoritama ovom problemu daje novu stranu kroz uređivanje samog procesa dobivanja modela. Iako je primjena takvih algoritama otežana zbog vremenskog ograničenja, oni su primjenjivi, a razvojem tehnologije i ubrzavanjem računala takve metode također će uskoro postati znatno popularnije zbog njihove jednostavnosti i brojnih mogućnosti.

Literatura

[1] PCG WIKI, Algorithms for Procedural Content Generation, http://pcg.wikidot.com/category-pcg-algorithms, travanj 2014.

[2] DOULL, A. The Death of a Level Designer, http://roguelikedeveloper.blogspot.com/2008/01/death-of-level-designer-procedural.html, travanj 2014.

[3] WIKIPEDIA, Procedural Generation, http://en.wikipedia.org/wiki/Procedural_generation, travanj 2014.

[4] TOGELIUS, J., YANNAKAKIS, G. N., STANLEY, K. O., BROWNE, C. Search-based Procedural Content Generation, Applications of Evolutionary Computation, 2010.

[5] GRUPA AUTORA. Procedural Content Generation in Games, http://pcgbook.com/, lipanj 2014.

[6] EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., WORLEY, S. *Texturing & Modeling: A Procedural Approach*, Third Edition

[7] ELIAS, H., Perlin Noise, http://freespace.virgin.net/hugo.elias/models/m_perlin.htm, svibanj 2014.

[8] PERLIN, K. Making Noise, http://www.noisemachine.com/talk1/, svibanj 2014.

[9] ZUCKER, M., The Perlin Noise Math FAQ, http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html, svibanj 2014.

[10] PERLIN, K. Noise and Turbluence, http://mrl.nyu.edu/~perlin/doc/oscar.html, svibanj 2014.

[11] KUSHNER, A., VYACHESLAV, S. Procedural Content Generation for Real-Time 3D Applications, http://unigine.com/articles/130605-procedural-content-generation/, svibanj 2014.

[12] BENTLEY, P. J. et al. *Evolutionary design by computers*, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1999.

[13] MARTZ, P. Generating Random Fractal Terrain, http://www.gameprogrammer.com/fractal.html, lipanj 2014.

[14] SARAFOPOULOS, A., BUXTON, B. F. Evolutionary algorithms in modeling and animation, Handbook of computer animation, Springer-Verlag, London, 2003.

[15] ALSING, R. Genetic Programming: Evolution of Mona Lisa, http://rogeralsing.com/2008/12/07/genetic-programming-evolution-of-mona-lisa/, lipanj 2014.

[16] COLLINGRIDGE, P. Evolving Images, http://www.petercollingridge.co.uk/book/export/html/14, lipanj 2014.

[17] BE-RAD, Voxelizing A Mesh in Unity, http://www.be-rad.com/2013/03/24/voxelizing-a-mesh-in-unity/, lipanj 2014.

[18] ROSEN, D. Triangle mesh voxelization, http://blog.wolfire.com/2009/11/Triangle-mesh-voxelization, lipanj 2014.

[19] NIHILIST DEV, Voxelization in Unity, http://nihilistdev.blogspot.com/2012/08/voxelization-in-unity.html, lipanj 2014.

[20] ALAIN, F. How to make a voxel based game in Unity?, http://www.sauropodstudio.com/how-to-make-a-voxel-base-game-in-unity/, lipanj 2014.

[21] KALOGERAKIS, E., CHAUDHURI, S., KOLLER, D., KOLTUN, V. A probabilistic Model for Component-Based Shape Synthesis, ACM Transactions on Graphics 31(4), 2012.

[22] MERELL, P., MANOCHA, D. Model Synthesis: A General Procedural Modeling Algorithm, IEEE transactions on visualization and computer graphics.

[23] MERREL, P. Example-Based Model Synthesis, Proceedings of the 2007 Symposium on Interactive 3D graphics and games.

Sažetak

PROCEDURALNO GENERIRANJE GRAFIČKIH OBJEKATA

Sažetak

Kroz rad opisane su i obrađene neke osnovne metode proceduralnog generiranja grafičkih objekata. Implementirane su metode potpunog generiranja sadržaja kroz kod, generiranja dvodimenzionalnog i trodimenzionalnog sadržaja korištenjem evolucijskih algoritama, te sinteza modela iz raznih predložaka. Opisane su razne varijacije tih metoda te neke srodne metode, a za svaku implementaciju opisane su prednosti i mane. Napravljena je usporedba u brzini dobivanja sadržaja između genetskog algoritma i vokseliziranja objekata za potrebe stvaranja trodimenzionalnih objekata.

Ključne riječi: računalna grafika, proceduralno generiranje, genetski algoritam, voksel, sinteza modela

PROCEDURAL GENERATION OF GRAPHICAL OBJECTS

Abstract

Various methods used in procedural generation of graphical objects were described and used through the work. The implementation includes methods used for generating content entirely through code, generating content using evolutionary algorithms in two and three dimensions and model synthesis from various templates. Some variations of these methods and similar methods have also been described, including a list of advantages and disadvantages for every part of the implementation. A comparison in performance between the genetic algorithm and the process of object voxelization for the purposes of creating three dimensional content was made.

Keywords: computer graphics, procedural generation, genetic algorithm, voxel, model synthesis