PROCEDURALNO GENERIRANJE TEKSTURA

Algoritmi i primjene

Ivan Galić
0036412475
Kolovoz 2011.

Mentorica: prof. Željka Mihajlović
Fakultet elektrotehnike i računarstva, Zagreb

SADRŽAJ

1. UVOD

1.1 Što su proceduralne teksture i pregled razvoja

U računalnoj grafici, teksture su jedan od osnovnih građevnih elemenata. Najčešće se koriste za oblaganje 3D objekata kako bi se uz što manji broj poligona održala razina detalja. Tipično, teksture se kreiraju iz fotografija koje se obrađuju i kombiniraju, ili crtanjem - ali opet direktnim mijenjanjem slikovnih elemenata i ručnim dodavanjem efekata i objekata na teksturu.

Još u samim počecima računalne grafike, razvila se ideja da bi se neke teksture mogle kreirati matematičkim putem, gdje bi se umjesto direktnog upravljanja slikovnim elementima mogle opisati raznim matematičkim funkcijama i odgovarajućim parametrima iz čega bi onda računalo generiralo teksturu (odnosno konačne slikovne elemente).

Glavna prednost ovog pristupa bila je ušteda na memoriji i neka svojstva koje rasterske teksture nemaju. Naime, u vrijeme kad je memorija (i RAM i disk prostor) bila veoma ograničena, spremanje tekstura u punoj rezoluciji kao raster slike predstavljao je problem. Nasuprot tome, za proceduralnu teksturu dovoljno je spremiti niz operacija koje je potrebno izvršiti i njihove parametre, da bi se tekstura mogla generirati u trenutku kad je zapravo potrebna. Nadalje, proceduralnu teksturu moguće je generirati u bilo kojoj rezoluciji pa je ona prilagodljiva s obzirom na dostupnu snagu računala na kojem se aplikacija izvodi. Osim toga, relativno je jednostavno dobiti teksture koje se mogu koristiti pri popločavanju, moguće je izbjeći probleme koji nastaju pri smanjivanju tekstura itd.

Međutim, prvi pokušaji proceduralnog generiranja tekstura davali su rezultate koji su izgledali umjetno, mogao se dobiti samo veoma uzak spektar različitih tipova tekstura i sama generacija je bila veoma spora osim za najjednostavnije slučajeve.

Tako su proceduralne teksture ostale u domeni istraživanja s relativno malom primjenom u praksi, osobito kod iscrtavanja u stvarnom vremenu, sve dok primjenu za njih nisu našle tzv. “demo grupe”; timovi koji su se natjecali u kreiranju što ljepše i tehnički impresivnije animacije u stvarnom vremenu, “demoa”, najčešće uz zadana ograničenja. Možda najpoznatiji takav oblik animacije je čuveni “64k demo”, gdje je ograničenje da cijeli izvršni program mora stati u samo 64kb memorije na disku. Očito, u tako malo prostora ne stane puno raster tekstura, 3d modela, glazbe i ostalih komponenti koje su potrebne za animaciju. Ovi timovi zato su odlučili koristiti proceduralno generirane teksture dovodeći tehnologiju njihove izrade i generacije na sasvim novu razinu. Uveli su nove algoritme za generaciju, kombiniranje i obradu tekstura

(programski), ali i razvili alate koji su omogućili grafičarima da stvore impresivne uratke i zatim ih spreme u svega nekoliko desetaka byteova.

Iz ove demo scene izašlo je i nekoliko komercijalnih alata zajedno sa skupom programskih knjižnica za generiranje tekstura, koji su tehnologiju ponovno digli na još višu razinu, pa su se tako teksture mogle generirati u gotovo stvarnom vremenu, koristeći grafičku karticu te razdvojiti na potrebne komponente u 3d grafici (npr. bump mapu, normal mapu, displacement mapu...). Možda još važnije, tvrtke koje su razvijale ove alate su ih promovirale tvrdeći da je puno brže grafičarima kreirati proceduralne teksture nego ih izrađivati u Photoshopu kao što je uobičajeno.

Kada su se pojavile i grafičke kartice s podrškom za programabilno sjenčanje, brzina kojom su se teksture mogle generirati se višestruko povećala, a tada su se počele koristiti u igrama, posebno onima koje su bile distribuirane digitalno i gdje je veličina ukupne igre bila važna (pa je tako poznata igra RoboBlitz koja je sve teksture generirala u realnom vremenu). Ipak, s vremenom je ušteda na prostoru izgubila na važnosti i danas se proceduralne teksture većinom koriste u sklopu paketa za 3D modeliranje i animaciju, pogotovo za izradu organskih tekstura (kože, pokrova bilja i sl.) jer je uporabom modernih algoritama takve teksture lakše dobiti na ovaj način nego iz fotografija.

Ciljevi ovog rada su prikazati neke od metoda i algoritama generacije proceduralnih tekstura, primjere ostvarivih rezultata te naposljetku pokazati implementaciju koja se može izvoditi na grafičkoj kartici i usporediti brzinu izvođenja u odnosu na implementaciju koja se izvodi na centralnom procesoru.

Slika 1. Primjeri proceduralno generiranih tekstura (prve dvije lijevo iz 64k FPS demoa, ostale iz programa uz ovaj rad)

1.2 Prednosti i nedostaci proceduralnih tekstura

Neke prednosti i nedostatke (u odnosu na raster teksture) sam već naveo u uvodu, no evo i sažeti pregled.

Prednosti

1. Mogućnost generiranja u bilo kojoj rezoluciji (pa ju aplikacija može prilagoditi dostupnim sklopovskim mogućnostima trenutnog računala)
2. Jednostavna prilagodba tekstura za potrebe popločavanja (svi algoritmi za proceduralno generiranje mogu generirati teksture koje se mogu popločavati bez većih problema)
3. Jednostavna promjena bez potrebe za ponovnim crtanjem cijele teksture što ubrzava razvoj (nakon promjene neke od matematičkih komponenti ili nekog od parametara, dovoljno je jednostavno regenerirati teksturu)
4. Jednostavno generiranje uzoraka koje je teško dobiti iz fotografija ili crtanjem (npr. koža vanzemaljske životinje)
5. Malo zauzeće memorije (dovoljno je spremiti upute za generiranje teksture)

Nedostaci

1. Neke teksture je teško ili nemoguće dobiti na ovaj način - najvažniji razlog (što se može ili ne može dobiti uvelike ovisi o kvaliteti alata odnosno algoritmima koji s dostupni)
2. Gubitak umjetničke slobode (grafičari imaju puno manju kontrolu nad matematičkim algoritmima koji generiraju teksture nego pri direktnoj manipulaciji slikovnim elementima)
3. Generacija tekstura dodatno traje i ovisi o složenosti teksture (za razliku od raster tekstura gdje vrijeme učitavanja ne ovisi o složenosti)
4. Dostupnost i cijena alata u usporedbi s rasterskim (razvoj je skup, postojeći alati isto tako)

2. OPENCL

2.1. OpenCL i proceduralne teksture

Kao što sam napomenuo u uvodu, dat ću usporedbu brzine generacije tekstura na glavnoj procesorskoj jedinici i na grafičkoj kartici. Kako se generacija proceduralnih tekstura može uvelike paralelizirati, odličan je kandidat za izvršavanje na grafičkoj kartici pa sam odlučio provjeriti koliko je zapravo ubrzanje (i postoji li uopće) te pojavljuju li se pri tome neočekivane poteškoće.

Za obje metode generacije morao sam odabrati neku od dostupnih tehnologija; C++ se nametnuo kao logičan izbor za prvu zbog svoje brzine, no za drugu metodu sam imao na izbor nekoliko tehnologija koje se po brzini u osnovi ne razlikuju ali se razlikuju po mogućnostima.

Za OpenCL sam se odlučio primarno zbog portabilnosti; OpenCL implementacije trenutno postoje za sve važnije platforme i za sve važnije (i novije) grafičke kartice.

Pri opisu implementacije pojedinih algoritama navest ću konkretne probleme s kojima sam se susreo, ali prije toga htio bih navesti neka ograničenja i pravila koja vrijede za sve OpenCL programe.

2.2 Arhitektura OpenCL-a i način primjene

Glavni razlog velike brzine grafičkih kartica su iznimna paralelizacija na sklopovskoj razini i protočna struktura. Uzrok tome leži u osnovnoj namjeni grafičkih kartica - naime, u 3D grafici računanje mnogih elemenata može se izvesti potpuno neovisno bilo da se radi o obradi vrhova trokuta ili pojedinih piksela, pa je i sklopovska arhitektura grafičkih kartica bila tome prilagođena još od samih početaka.

Međutim, upravo zbog takve arhitekture grafičke kartice su veoma neprilagođene i spore u izvođenju zadataka koji se ne mogu dovoljno paralelizirati, jer tada veći dio sklopovlja na grafičkoj kartici besposleno čeka. Ostala ograničenja su općenito manji broj registara od centralnog procesora, manja količina memorije te ograničena propusnost prema centralnom procesoru odnosno središnjoj memoriji, a posebno ulazno-izlaznim jedinicama. Kao ograničenje samog jezika vrijedno je spomenuti i nedostatak rekurzije - naime rekurzivno pozivanje funkcija nije dopušteno prema trenutnoj specifikaciji (premda je za očekivati da će s vremenom to ograničenje nestati).

Iz navedenih razloga, OpenCL i slični jezici trenutno nisu primjenjivi na širok spektar problema kao što je to slučaj s kodom koji se izvršava na centralnoj upravljačkoj jedinici, ali za probleme koji zahtijevaju intenzivne numeričke proračune koji se mogu izvoditi paralelno (kao što je slučaj s proceduralnim teksturama) mogu dati velika ubrzanja.

Ovdje bih još napomenuo da iako sam spominjao OpenCL samo u kontekstu izvođenja na grafičkoj kartici, on zapravo nije ograničen na određeni tip sklopovlja, pa tako postoje i implementacije za centralnu procesorsku jedinicu i neke druge procesore posebne namjene (npr. IBM Cell Blade). Izvođenje na centralnoj jedinici ipak nema puno smisla u kontekstu generacije proceduralnih tekstura (a specijaliziranim ubrzivačkim jedinicama nisam imao pristup) pa sam ovdje razmatrao samo izvođenje na grafičkim karticama.

2.3 Implementacija algoritama u OpenCL-u

Implementacija ovih algoritama u OpenCL-u je u većem dijelu direktno prenesena iz ekvivalentne C implementacije za centralni procesor, pri čemu je glavni dio postupka izdvajanje dijela koda koji se izvršava za svaki piksel i njegovo smještanje u tzv. "kernel". Kernel je posebna funkcija unutar OpenCL programa koju pozivamo iz programa "domaćina" (engl. host; program koji se izvodi na centralnoj jedinici, u našem slučaju pisan u C++). U pravilu, kernel izračunava vrijednost jednog piksela, pa se iz programa domaćina postavke urede na takav način da se taj kernel pozove NxN puta, gdje je N širina i visina teksture (pod pretpostavkom da je tekstura kvadratna). Kernel pomoću ugrađenih funkcija može dohvatiti koordinatu piksela koji trenutno obrađuje pa s obzirom na nju vrši izračun i rezultat upisuje u izlazni međuspremnik (buffer). Na ovaj način - instanciranjem NxN kernela - postižemo maksimalnu paralelizaciju i brzinu. Kada cijeli niz algoritama završi s radom (dakle imamo konačnu teksturu), vrši se prijenos izlaznog međuspremnika u sistemsku memoriju iz koje se onda kreira OpenGL tekstura.

Moram napomenuti da ovaj posljednji korak nije nužan; naime kako se izlazni međuspremnik već nalazi u video memoriji gdje treba biti i OpenGL tekstura, moguće je povezati OpenGL i OpenCL na takav način da se taj izlazni međuspremnik odmah koristi kao OpenGL tekstura pri čemu onda izbjegavamo ovo kružno putovanje od video memorije u sistemsku i natrag. Nažalost, ova funkcionalnost je usko vezana uz platformu jer je potrebno povezati OpenCL kontekst (koji je definiran specifikacijom i kreira se na isti način na svim paltformama) i OpenGL kontekst (koji nije definiran specifikacijom pa je na svakoj platformi predstavljen na drukčiji način). Premda je ovaj dvostruki prijenos podataka preko sabirnice spor, u ovom slučaju ne utječe bitno na rezultate pa sam se odlučio za tu varijantu koja je prenosiva (neovisna o platformi).

Radi potpunosti, slijedi kratki pregled postupka povezivanja OpenCL i OpenGL sustava.

Prvi korak je povezati OpenGL kontekst s OpenCL kontekstom, što ovisi o operacijskom sustavu pod kojim se izvodi pa ga ovdje neću detaljnije obrađivati.

Srećom, ostali koraci su definirani OpenCL specifikacijom i rade na svim platformama jednako. Počinjemo kreiranjem OpenGL teksture na standardni način, a zatim pomoću funkcije *clCreateFromGLTexture2D* dobijamo OpenCL memorijski objekt.

Kada želimo pristupati toj OpenGL teksturi, moramo "zaključati" pripadni memorijski objekt pozivom funkcije *clEnqueueAcquireGLObjects*, nakon čega je on spreman za korištenje u kernelima. Nakon što smo završili obradu (dakle, nakon što smo izvršili potrebne kernele) moramo "otključati" teksturu kako bi ju OpenGL mogao koristiti. Ovaj slijed - "zaključaj", obradi, "otključaj" možemo ponavljati proizvoljan broj puta.

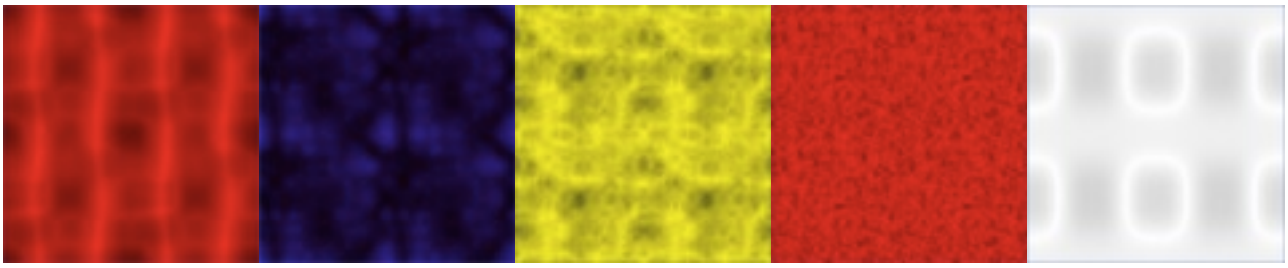
3. POSTUPCI GENERACIJE

3.1 Načini dobijanja proceduralnih tekstura

Osnova za dobijanje proceduralnih tekstura su generacijski algoritmi, odnosno algoritmi koji kao ulaz primaju samo skup parametara a kao izlaz daju skup slikovnih elemenata odnosno teksturu. No, generacijski algoritmi kreiraju teksture njima specifičnog izgleda, zbog čega sami po sebi ne pružaju dovoljnu raznolikost. Na primjer, Turbulence generira teksture koje liče na oblake ili dim dok Worley noise generira teksture sa staničnim uzorkom. Zato se pri kreiranju proceduralne teksture obično koristi nekoliko algoritama od kojih neki mogu biti generacijski, a neki manipulatorski (koji na neki način mijenjaju postojeću teksturu) te se njihovim kombiniranjem postiže željeni rezultat. Za ovaj rad implementirao sam nekoliko generacijskih i nekoliko manipulativnih algoritama koje ću detaljnije opisati u nastavku.

3.2 Algoritam Turbulence

Kao što sam spomenuo, ovaj algoritam generira teksture koje većinom liče na oblake, dim, tekućinu ili nešto slično. Evo nekoliko primjera:



Slika 2. Primjeri tekstura generiranih programom uz ovaj rad.

Na slikama možemo vidjeti nekoliko važnih osobina ovog generatora. Sve teksture se mogu spajati sa svih strana bez vidljivih spojeva. U ovom slučaju, svih 5 tekstura su zapravo 2x2 uzorci originalne teksture. Uočljive su i različite razine zrnatosti, pa su tako slike 1 i 5 veoma glatke, dok se na slikama 2 i 3 mogu vidjeti sitni detalji koji liče na zrnca pijeska posutog po teksturi. Premda i same po sebi zanimljive, u rijetkim se slučajevima mogu iskoristiti kao takve; umjesto toga se najčešće koriste kao baza za teksturu ili filtar pri čemu rezultatu daju prirodniji izgled.

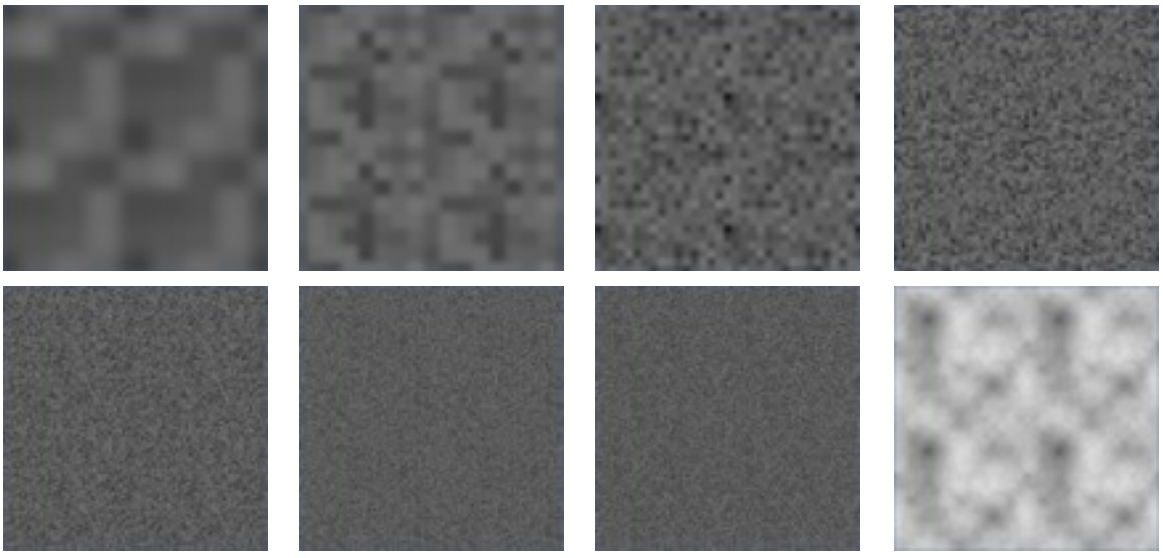
Algoritam

Osnovni pristup

Način rada algoritma najbolje možemo shvatiti proučavajući pridodne fenomene: naime, gledajući planine, valove na vodi, oblake ili šare u mramoru, možemo primijetiti različite razine detalja.

Uzmimo za primjer oblake. U najkrupnijem planu, vidimo njihove obrise kako ocrtavaju oblike oblaka na nebu. Pozornijim promatranjem, možemo primijetiti da rubovi zapravo nisu sasvim glatki nego imaju nepravilnosti. Ako bismo ih mogli pogledati još bliže, vidjeli bismo sve sitnije i sitnije nepravilnosti, kako na rubovima tako i u unutrašnjosti oblaka. Jedna od ključnih stvari za ovaj algoritam je i kako sve sitniji, nepravilniji detalji imaju sve manji utjecaj. Izdaleka, oblaci izgledaju sasvim glatko, dok se nepravilnosti mogu primijetiti tek iz veće blizine. Slična je stvar s obrisima planina, pri čemu izdaleka vidimo samo najveće, dok kad se približavamo možemo vidjeti sve manje planine, pa brda, pa zatim nepravilne obrise brda, pa stijene, njihove nepravilne obrise, kamenčiće itd.

Na ovoj ideji je baziran i Turbulence algoritam. Generira se zadani broj slojeva, tzv. oktava, pri čemu je svaki sljedeći sve detaljniji. Oni se na kraju zbrajaju, pri čemu slojevi s najviše detalja imaju najmanji utjecaj na finalnu sliku, a slojevi s najmanje detalja najveći. Promotrimo sedam oktava jedne teksture i finalni rezultat:



Slika 3. Sedam oktava jedne teksture i finalni rezultat

S lijeva na desno, od gore prema dolje, prikazano je sedam oktava od najmanje razine detalja do najveće, te na kraju konačni rezultat koji je zbroj svih sedam oktava (ponovno, uzimajući u obzir da prva oktava ima najveći utjecaj, a sedma najmanji).

POSTUPAK

Turbulence algoritam može se koristiti u bilo kojem broju dimenzija, a ja ću ovdje pokazati 2D inačicu jer se ona koristi za generaciju 2D tekstura. Parametri algoritma su broj oktava (slojeva) koje želimo generirati, početna razina detalja (zoom), te dvije boje između kojih algoritam interpolira rezultate. Zoom je parametar koji predstavlja najnižu (najgrublju) razinu detalja, a algoritam za svaku sljedeću oktavu generira duplo višu razinu detalja (Slika 3).

Osnova algoritma je uniformni random šum veličine teksture koju želimo generirati. Ovo će nam biti baza za sve slojeve (oktave), jer da bismo pri zbrajanju oktava dobili rezultat bez prekida, moramo uzorkovati iste vrijednosti. Šum generiramo kao decimalne brojeve u vrijednosti od 0 do 1 koristeći random funkciju. Rezultat je sličan kao na slici 3, sedma podslika.

Rezultat računanja svake oktave je skup slikovnih elemenata veličine ciljne teksture, a u osnovi ga dobijamo interpoliranim povećavanjem podskupa izvornog šuma. Objasnit ću na pojednostavljenom pseudokodu (kod prikazuje računanje jednog slikovnog elementa jedne oktave na lokaciji i,j):

```
// podrazumijevamo da je originalni šum generiran u noise[width][height]

// currentOctave je oktava koju trenutno računamo

// startingSampleRate je učestalost uzorkovanja na prvoj oktavi, računa se kao zoom * width

float sampleRate = startingSampleRate / 2^currentOctave;

float fractionalX = i / sampleRate;

float fractionalY = j / sampleRate;

int pixelX = (int)fractionalX;

int pixelY = (int)fractionalY;

float dx = fractionalX - pixelX;

float dy = fractionalY - pixelY;

pixelX %= width; pixelY %= height;

float n1 = Interpolate(noise[pixelX, pixelY],

                       noise[(pixelX + sampleRate) % width, pixelY],
```

```
                        dx);
float n2 = Interpolate(noise[pixelX, (pixelY + sampleRate) % height],
                        noise[(pixelX + sampleRate) % width, (pixelY + sampleRate) % height],
                        dx);
float result = Interpolate(n1, n2, dy);
```

Prvi korak je izračunati učestalost uzorkovanja na trenutnoj oktavi. Za primjer, uzmimo da je parametar zoom = 0.25 i veličina teksture 256. Tada je startingSampleRate = 0.25 * 256 = 64. Nadalje, sampleRate za prvu oktavu je 64, za drugu 32, za treću 16 itd. Ovaj broj predstavlja svakih koliko slikovnih elemenata ćemo uzorkovati novu vrijednost iz originalne matrice noise.

Iz lokacije (i, j) računamo koji slikovni element uzorkujemo iz originalnog šuma (pixelX, pixelY) i koliko smo odmaknuti od tog slikovnog elementa prema idućem (dx, dy). Brojevi koji predstavljaju odmak će uvijek biti u intervalu [0, 1), gdje nula predstavlja da se nalazimo točno na trenutnom slikovnom elementu (pixelX, pixelY) bez odmaka, dok bi 1 predstavljao sljedeći slikovni element koji uzorkujemo (pixelX + sampleRate, pixelY + sampleRate).

Kako bismo dobili glatku sliku šuma, interpoliramo vrijednosti trenutnog slikovnog elementa u originalnoj matrici i sljedeća tri koja ćemo uzorkovati (desno, dolje i dolje-desno), s obzirom na udaljenosti (dx, dy) od trenutnog slikovnog elementa. Funkcija Interpolate može biti implementirana na razne načine, od kojih su najpopularniji linearna interpolacija (najbrži način), kubična (visoke kvalitete) i sinusoidalna (dobar omjer kvalitete i brzine). Što kvalitetniju funkciju interpolacije upotrijebimo, dobit ćemo zaglađeniju i prirodniju sliku. Interpolacija se računa u dva smjera - s lijeva na desno uz trenutni y, koristeći pomak dx, a zatim između te dvije vrijednosti koristeći pomak dy.

Kada znamo izračunati boju jednog slikovnog elementa u jednoj oktavi, izračunati finalnu boju slikovnog elementa kao zbroj oktava je jednostavno. Kao što sam spomenuo gore, svaka oktava ulazi u finalni rezultat s određenom jačinom, koja se zove amplituda. Postoje razni načini računanja amplitude ali univerzalno najbolje rezultate daje postupak u kojem prva oktava ima amplitudu jedan, a svaka sljedeća duplo manju. Ako za primjer uzmemo teksturu s tri oktave čiji su rezultati za dani slikovni element oct1, oct2 i oct3, finalni rezultat za taj slikovni element bismo izračunali ovako:

$$result = oct1 * 1.0 + oct2 * 0.5 + oct3 * 0.25 \quad (1)$$

Nastavljajući gornji primjer, brzo bismo došli do situacije gdje nove oktave više nemaju vidljivog utjecaja na teksturu, jer je amplituda manja od razlučivosti jednog kanala boje (1 / 256 za 8bit/channel).

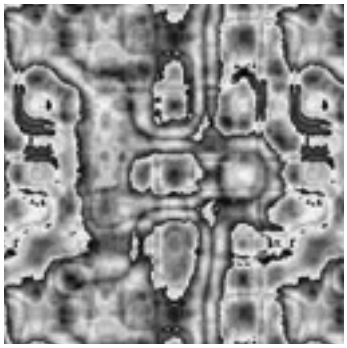
Ponavljajući ovaj postupak za svaki slikovni element u teksturi onoliko puta koliko ima oktava, dobijamo konačni rezultat. Pogledajmo sada neke detalje implementacije i utjecaj parametara na izgled slike.

DETALJI IMPLEMENTACIJE

Implementaciji Turbulence algoritma moguće je pristupiti na više načina, a ja sam odabrao onaj gdje se prvo generira šum veličine teksture (radi postizanja veće brzine), a zatim se računa finalna vrijednost svakog slikovnog elementa kroz sve oktave:

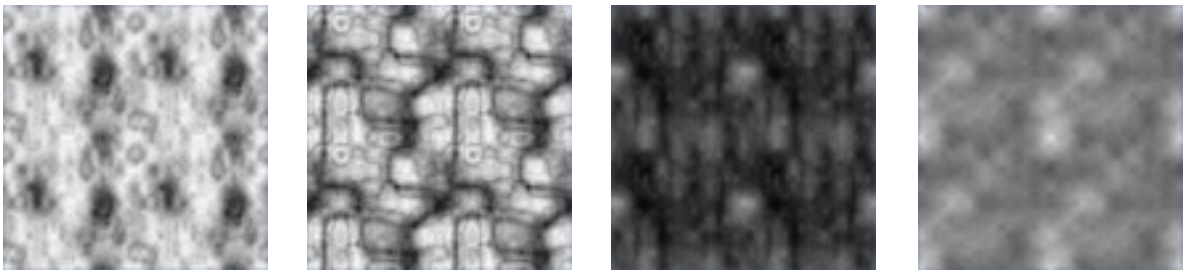
```
for i = 1 to width
        for j = 1 to height
                val = 0
                amplitude = 1.0
                for k = 1 to numOctaves
                        izracunaj sample rate
                        izracunaj result = vrijednost slikovnog elementa za trenutnu oktavu
                        val += result * amplitude
                        amplitude = amplitude * 0.5
                end
                pixel[i, j] = color1 * (1 - val) + color2 * val
        end
end
```

Osim toga, algoritam ima nekoliko ključnih točaka gdje se može utjecati na izlaz algoritma. Šum koji se generira na samom početku algoritma i koristi u svim daljnjim koracima moguće je ostaviti kakav je ili ga se može "zagladiti" primjenom neke vrste algoritma za zamućivanje. Ukoliko ga se ostavi u originalnoj formi, dobija se oštrija slika na izlazu, s više detalja, ali se također puno češće pojavljuju problemi gdje zbog naglih promjena u vrijednostima izvornog šuma dolazi i do naglih promjena u izlaznoj slici, što ružno izgleda.



Slika 4. Odsijecanje

Sljedeće mjesto gdje se promjenom parametara mogu dobiti zanimljivi rezultati je interpolacija. Sjetimo se, interpolacija se za svaki slikovni element vrši tri puta u samom algoritmu, te još jednom pri određivanju finalne boje. Evo nekoliko primjera različitih rezultata koje možemo dobiti mijenjajući načine interpolacije:



Slika 5. Različiti načini interpolacije vrijednosti

Na slici 5, korišteni su isti parametri algoritma (zoom, broj oktava, veličina tekstura, boje), ali su vrijednosti n1, n2 i konačni rezultati interpolirani na različiti način. Na primjer:

```
n1 = sin(3.1415926 / 2 * Abs(1 - 2 * Interpolate(noise[pxi][pyi],
                                                 noise[(pxi + sampleRate) % width][pyi],
                                                 dxf)));
n2 = sin(3.1415926 / 2 * Abs(1 - 2 * Interpolate(noise[pxi][(pyi + sampleRate) % height)],
                                                 noise[(pxi + sampleRate) % noiseW + ((pyi +
sampleRate) % noiseH)],
                                                 dxf)));
val = Interpolate(n1, n2, dyf) * amplitude;
```

Ovdje funkcija Interpolate radi kubičnu interpolaciju, a rezultat je treći primjer na slici 5.

Konačno, mijenjanjem načina kombiniranja boja, posebno upotrebom gradijenata, možemo dobiti još zanimljivije rezultate, povećati ili smanjiti kontrast i sl. Međutim, to se najčešće ne radi u sklopu Turbulence algoritma nego kao poseban korak pri generaciji proceduralnih tekstura.

OpenCL specifičnosti

U C++ implementaciji se isplati generirati međuspremnik s originalnim šumom kako bi se kasnije izbjeglo računanje i razlika u brzini je vidljiva. U OpenCL implementaciji zbog načina na koji radi protočna struktura nije uvijek jasno koji pristup će dati bolje rezultate. Ako šum generiramo u međuspremnik, potrebna su nam dva prolaza (dakle, dva kernela od kojih prvi generira čitavi osnovni šum, a nakon njega drugi generira konačnu teksturu) ali nema potrebe za računanjem random vrijednosti u drugom kernelu. Međutim, nije neobična ni varijanta u kojoj umjesto prvog koraka u kojem generiramo šum, jednostavno u drugom (i sada jedinom) kernelu izračunamo vrijednost random funkcije kada nam zatreba.

Odlučio sam isprobati obje varijante i premda su dale veoma slične rezultate (razlika u brzini generacije je varirala oko 0.1-0.5ms pri ukupnom vremenu od 2.8-3.3ms za teksturu 512x512 piksela, ovisno o parametrima), na moje iznenađenje druga varijanta (bez pripremnog koraka) je bila brža pa sam ju odlučio ostaviti u konačnoj implementaciji.

Dakle, svaki put kada je potrebna random vrijednost, poziva se random funkcija. No, kako se pri računanju različitih oktava (pa i različitih piksela iste oktave) moraju koristiti iste vrijednosti izvornog šuma (inače dobijamo nagle skokove u vrijednostima), potrebna je i drukčija random funkcija.

Tipično, random funkciji na početku programa predajemo inicijalni broj (ili brojeve), tzv. "seed", a zatim uzastopnim pozivanjem funkcije dobijamo različite ravnomjerno raspoređene vrijednosti, pri čemu funkcija interno mijenja stanje svojih varijabli i čuva njihovu vrijednost. Međutim, u ovom algoritmu ćemo više puta morati dohvatiti vrijednost originalnog šuma na istoj poziciji u različitim, neovisnim pozivima nekog kernela pri čemu očekujemo dobiti isti rezultat. Zato nam je potrebna random funkcija koja ne ovisi o internom stanju, nego kao parametre ima koordinate i inicijalni broj (seed), te za uređenu trojku (x, y, seed) uvijek daje isti rezultat. Na taj način, ako sve kernele pozovemo s istim inicijalnim brojem kao parametrom (njega generiramo u programu domaćinu), moći ćemo uvijek dohvatiti istu vrijednost originalnog šuma u istoj točki, a zadržavamo mogućnost generiranja različitih tekstura jer svaki puta možemo poslati drugi inicijalni broj kernelima pri generaciji.

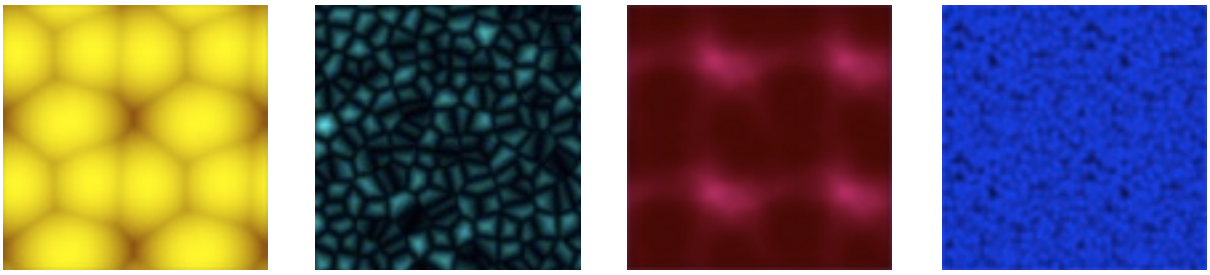
Nakon isprobavanja različitih algoritama za generiranje random brojeva i njihove modifikacije da primaju uređenu trojku kao parametre odlučio sam se za kombinaciju unaprijed generiranog niza slučajnih brojeva po uzoru na implementaciju iz [1] u koji se indeksiram pomoću zadane trojke, te zatim računanje potencije dobivenih rezultata:

```
slucajni_niz[] = { ... } // Duljine LEN

int random(int x, int y, int z)

{

    x = slucajni_niz[( (x mod LEN) + slucajni_niz(y mod LEN) ) mod LEN];

    y = slucajni_niz[( (y mod LEN) + slucajni_niz(z mod LEN) ) mod LEN];

    return x ^ y;

}
```

Za druge algoritme bio mi je potreban i algoritam koji radi na "klasičan" način, dakle uz čuvanje stanja i neovisno o koordinatama, što ću pokazati u nastavku.

3.3 Algoritam Worley noise

Drugi generativni algoritam koji sam ovdje prikazao je tzv. Worley noise, kojeg je osmislio Steven Worley 1996. godine. Daje rezultate koji najviše podsjećaju na staničnu strukturu ili nakupine bakterija, pa se zbog toga najčešće i upotrebljava kao baza za teksture sličnog tipa (npr. koža, pčelinje saće i sl.). Evo nekoliko primjera iz programa:



Slika 6. Worley teksture

ALGORITAM

Konceptualno, ovaj algoritam je jednostavan, a sastoji se od sljedećih koraka:

1. Generiraj N lokacija P(Xi, Yi) na teksturi
2. Za svaki slikovni element(i, j), izračunaj minimalne udaljenosti Min1(i, j) i Min2(i, j) do dvije najbliže točke P
3. Izračunaj vrijednost slikovnog elementa u točki (i, j) koristeći vrijednosti Min1(i, j) i Min2(i, j), npr.:
    a. vrijednost = Min1(i, j) - primjer 1 na slici 6
    b. vrijednost = Min2(i, j) - Min1(i, j) - primjer 2 na slici 6
    c. vrijednost = Min1(i, j) * Min2(i, j) - primjer 3 na slici 6
    d. vrijednost = Min2(i, j) - primjer 4 na slici 6

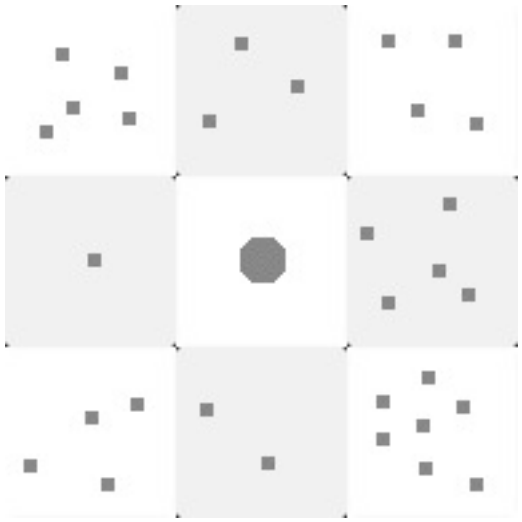
Na izgled konačnog rezultata utječe način generacije inicijalnih točaka P (način distribucije) i njihov broj, te način računanja vrijednosti slikovnog elementa (korak 3), kao što možemo vidjeti

iz priložene slike. Točke P u finalnoj slici predstavljaju centre vizualno razlučivih područja u konačnoj teksturi, koja podsjećaju na staničnu strukturu.

Detalji implementacije

Premda konceptualno jednostavan, implementacija ovog algoritma može biti problem ako želimo postići dobru brzinu generacije. Naime, računanje vrijednosti Min1 i Min2 za svaki slikovni element može biti veoma sporo, posebno za velike teksture (moja prva implementacija trajala je toliko dugo da sam mislio da se algoritam zaglavio).

Odlučio sam se za implementaciju u kojoj teksturu razdijelim na kvadrataste ćelije što olakšava nalaženje najbližih P točaka. Naime, ako promatramo neki slikovni element u ćeliji (i, j), njemu najbliže točke P mogu se nalaziti samo u ćeliji (i, j) ili jednoj od 8 okolnih ćelija (ova tvrdnja vrijedi samo pod uvjetom da svaka ćelija sadrži najmanje jednu točku P). Ovisno o gustoći točaka P u teksturi, teksturu možemo podijeliti na manje ili veće ćelije, čime kontroliramo minimalne razmake među P točkama (oni nikada ne mogu biti veći od duljine jedne ćelije) kao i brzinu algoritma, jer pri ispitivanju pojedinog slikovnog elementa u ćeliji ispitujemo onoliko točaka koliko ih se nalazi u danih 9 ćelija oko tog slikovnog elementa.



Slika 7. centri ćelija u Worleyevom algoritmu

Dakle, implementacija pri generaciji P točaka svaku od njih smješta u pripadnu ćeliju radi lakšeg pretraživanja pri traženju minimalnih udaljenosti za pojedini slikovni element.

Nakon što su pronađene najmanje udaljenosti, vrijednost slikovnog elementa na toj lokaciji računamo nekom matematičkom funkcijom koja u obzir uzima jednu, drugu ili obje minimalne

udaljenosti (postoje i implementacije koje koriste 3 ili čak 4 najmanje udaljenosti u svojim kalkulacijama).

Gore sam naveo da je jedno od ograničenja implementacije da svaka ćelija u teksturi mora imati barem jednu P točku. Pokazalo se međutim da je ovo ograničenje većinom i poželjno pri generaciji tekstura, neovisno o algoritmu. Naime, bez njega generirane teksture imaju puno nepravilniji raspored P točaka pa cijela tekstura izgleda degenerirano (čime gubi izgled stanične strukture, pokrova kože, pčelinje saće i sl.).

Drugo ograničenje algoritma je da veličina ćelije mora biti djelitelj širine odnosno visine teksture (u suprotnom tekstura završava prije kraja zadnje ćelije pa se vide prijelazi).

OpenCL specifičnosti

Slično kao u Turbulence algoritmu i ovdje postupak možemo podijeliti na dva dijela, pri čemu se u prvom dijelu generiraju lokacije središta ćelija, a u drugom se izračunavaju udaljenosti do tih središta za svaki piksel i konačne vrijednosti piksela. Ponovno, moguće je ova dva koraka izvesti zasebno pa prvo generirati lokacije u međuspremnik, ili oba koraka spojiti u jedan pa umjesto generiranja lokacija u međuspremniku pozivati random funkciju za potrebne vrijednosti pri računanju svakog piksela.

Za razliku od Turbulence algoritma gdje je drugi pristup davao granično bolje rezultate, ovdje je prvi pristup bio brži (još više granično, oko 0.1-0.3ms pri ukupnom vremenu generiranja oko 2.7-3.0ms za teksturu veličine 512x512 piksela). Gotovo beznačajno, ali radi demonstracije obje tehnike ostavio sam drugi pristup s pred-popunjavanjem međuspremnika.

Implementirajući ovaj algoritam, naišao sam na još neke probleme specifične OpenCL-u. Kao vrlo mlada tehnologija, još uvijek postoje problemi s konkretnim implementacijama, pogotovo zato jer svaki proizvođač kreira svoju a i razlike u sklopovlju su velike.

Testirajući ovaj algoritam na MacBook Pro-u i grafičkoj kartici nVidia 8600GT zadnja tri retka teksture su uvijek ostajala crna (na drugoj konfiguraciji sve je radilo normalno). Pobližim proučavanjem ustanovio sam da se kerneli jednostavno ne pozivaju za ta dva zadnja retka, da bih na kraju pronašao da je problem u specificiranju duljine radne grupe. Naime, zbog bolje paralelizacije, kompletan prostor na kojem se izvršava algoritam (dakle NxN elemenata) dijeli se na manje blokove, radne grupe. OpenCL specifikacija predviđa mogućnost zahtjeva za automatskom podjelom na blokove, u kojem slučaju se pogonski programi brinu za optimalne postavke. Ja sam koristio tu mogućnost ali na spomenutoj grafičkoj kartici ona ne radi. Kada sam ručno dodijelio veličine radnih grupa, popunila su se i zadnja dva retka teksture.

Drugi problem se pojavio zbog moje greške, jer sam pristupao memoriji izvan granica međuspremnika. Za očekivati je da ovakav program neće raditi, ali na žalost pokretanje programa je uzrokovalo potpuno smrzavanje računala, što je otežavalo pronalaženje greške. OpenCL radi na jezgrenoj razini pa neke greške u programu mogu ugroziti stabilnost operacijskog sustava. Ovakvi problemi najvećim su dijelom posljedica kratkog vremena od kada je OpenCL dostupan pa je programska podrška još nestabilna.

3.4 Sinusni uzorak

Prošla dva opisana algoritma namijenjeni su generaciji nepravilnih uzoraka kako bi postigli što prirodniji izgled. Sinusni uzorak je prigodniji za pravilne strukture poput npr. rešetaka.

Osnovna ideja je kao intenzitet piksela u nekoj točki koristiti funkciju koja uključuje sinusnu funkciju, pri čemu kao parametre koristimo koordinate piksela.

Parametri funkcije su *xpower* i *ypower* koji u određuju kut pod kojim se uzorak generira i broj ponavljanja (kako bi se tekstura mogla nastavljati, parametri *xpower* i *ypower* moraju biti djelitelji duljine odnosno visine teksture). Osnovna funkcija izgleda ovako:

float val = sin((x / texture_width) * 2 * pi * xpower + (y / texture_height) * 2 * pi * ypower); (2)

Rezultat je ovog oblika:

Slika 8. Sinusni uzorak

Implementacija algoritma je jednostavno izračunavanje vrijednosti svakog piksela prema gornjoj formuli.

3.5 Bojanje gradijentom

Svi do sada navedeni algoritmi konačnu vrijednost preslikavaju između dvije dane boje. Često to nije dovoljno, nego je potrebno napraviti prijelaze između više različitih boja. Jednostavno rješenje je korištenje gradijenta, koji možemo zamisliti kao niz od 256 elemenata u kojem postoji N kontrolnih točaka kojima su pridijeljene boje. Za svaki od 256 elemenata, vrijednost gradijenta je interpolirana između boja u dvije kontrolne točke:

Slika 9. Primjer gradijenta

Na slici vidimo gradijent koji ima tri kontrolne točke: crnu na lokaciji 0, crvenu na lokaciji 128 (u sredini), te žutu na lokaciji 255.

Ovdje naziv "gradijent" koristim u značenju u kojem je poznat u korisničkim računalnim programima poput Adobe Photoshop-a, a ne u matematičkom značenju parcijalnih derivacija.

Slika 10. Primjer identičan gornjem samo obojan gradijentom

Prolazimo kroz sve piksele teksture, uzimamo vrijednost jednog od kanala (uzimao sam crveni kanal) i potražimo vrijednost gradijenta u toj točki. Tu boju koju vrati gradijent upisujemo kao konačnu boju teksture.

Implementacija u OpenCL-u je većinom jednaka jednostavnoj C implementaciji, osim što je kao dodatni parametar potrebno poslati i gradijent. To sam napravio tako da sam stvorio niz od 256 elemenata i u njega zapisao vrijednosti gradijenta u pojedinoj točki. Taj niz sam proslijedio OpenCL programu unutar kojega sam ga onda jednostavno indeksirao.

3.6 Distorzija pomoću druge teksture

Kako bismo dobili još prirodniji, nepravilniji uzorak, koristimo distorziju teksture pri čemu se druga tekstura koristi kao tablica koja određuje jačinu distorzije. Parametri su izvorna tekstura, tekstura koja služi kao tablica, parametri *xpower* i *ypower* koji određuju jačinu distorzije u pojedinom smjeru te izlazna tekstura, a za pojedini piksel vrijednost se računa ovako:

```
int sx = (x + tablica[x][y] * xpower) % texture_width

int sy = (y + tablica[x][y] * ypower) % texture_width

izlaz[x][y] = ulaz[sx][sy]
```

Ako generiramo teksturu Turbulence algoritmom pa ju primijenimo na teksturu od gore, dobit ćemo rezultat sličan ovome:

Slika 11. Gradijentom obojan sinusni uzorak, zatim distorziran pomoću teksture generirane Turbulence algoritmom

3.7 Multiplikativno miješanje tekstura

Konačno, različite zanimljive efekte možemo dobiti miješanjem dvaju tekstura. Postoje mnogi načini miješanja, a jedan od najkorisnijih je multiplikativni. Uz zadane teksture *texture1* i *texture2* postupak za izračunavanje piksela na lokaciji (x, y) izgleda ovako:

```
texture1[x][y] = to_color(uniform(texture1[x][y]) * uniform(texture2[x][y]))
```

Ovdje funkcija vrijednosti teksture koje su od [0,255] pretvara u interval [0, 1]. Nakon množenja rezultat ponovno pretvaramo u boju, odnosno množimo s 255 kako bismo dobili interval [0, 255].

Generirajući novu teksturu pomoću algoritma Turbulence te množeći s teksturom od gore, dobit ćemo rezultat sličan ovome:

Slika 12. Bojanje gradijentom sinusnog uzorka, distorzija pomoću teksture generirane Turbulence algoritmom i zatim miješanje s još jednom takvom teksturom.

3.8 Intenzitet

Ako nam je tekstura presvijetla ili pretamna, možemo podesiti intenzitet. Algoritam računa izlaznu vrijednost tako da jednostavno množi ulaznu vrijednost zadanim intenzitetom:

```
izlaz[x][y] = ulaz[x][y] * intenzitet
```

Ovdje treba obratiti pozornost na odsijecanje; ako množenjem dobijamo vrijednosti veće od 1 (nakon preslikavanja intervala [0, 255] na interval [0, 1]), rezultat će biti odsječen na vrijednost 1 što uništava glatke prijelaze na slici. Slična situacija se događa i kad množimo brojem manjim od 1 pa nam neki elementi teksture dobijaju vrijednost zaokruženu na nula.

3.9 Popločavanje

U opisu gornjih algoritama nisam se osobito osvrnuo na probleme nastavljanja tekstura (no primjeri generiraju teksture koje se mogu nastavljati), a ovo svojstvo je veoma važno zadovoljiti kako bi se teksture mogle koristiti u 3D grafici. Naime, pri korištenju tekstura u 3D grafici, često je potrebno jednu te istu teksturu ponoviti nekoliko puta, gdje se njena lijeva strana oslanja na desnu i/ili gornja na donju, a nužno je da se na tim prijelazima ne vide crte. Ovo je problem koji proceduralne teksture elegantno i jednostavno rješavaju, bez obzira o kojem se algoritmu radi.

Pri generaciji teksture, do prekida na rubovima će doći ukoliko se pri izračunavanju slikovnog elementa (i, j) koriste okolni slikovni elementi, pa onda na rubovima moramo koristiti neke fiksne vrijednosti (npr. uvijek prozirnu boju, bijelu, crnu ili možda boju slikovnog elementa na samom rubu). Međutim, kada se nakon toga spoje lijevi i desni, gornji i donji rub, vidljivi su nagli skokovi boje jer susjedni slikovni elementi nisu izračunati s istim okolnim vrijednostima.

Na sreću, ovaj problem je lako riješiti, i to tako da u svim slučajevima gdje se dohvaćaju vrijednosti okolnih slikovnih elemenata ne radimo provjere jesmo li došli do ruba teksture, nego pri računanju koordinata koristimo operator modulo s parametrom visine odnosno širine teksture. Na taj način će nam slikovni element desno od najdesnijeg slikovnog elementa u teksturi biti početni slikovni element lijeve strane teksture, a slikovni element ispod najdonjeg slikovnog elementa u teksturi bit će slikovni element sa samog vrha. Kao primjer, uzmimo liniju pseudokoda kojeg sam naveo pri opisu Turbulence algoritma:

```
float n1 = Interpolate(noise[pixelX, pixelY], noise[(pixelX + sampleRate) % width, pixelY], dx);
```

Na ovaj način svi slikovni elementi s rubova teksture će se slagati s nasuprotnim rubom pa neće biti naglih prijelaza i tekstura će biti pogodna za popločavanje.

4. SUČELJE PROGRAMA

Program je implementiran koristeći OpenGL, OpenCL i GLUT, i bez parametara sve algoritme izvodi koristeći OpenCL implementaciju. Ukoliko se kao parametar s komandne linije da "*nocl*", svi algoritmi izvodit će se u standardnoj C/C++ implementaciji.

Pokretanjem programa, prikazat će se prva generirana tekstura na poznatom modelu čajnika. Pritiskom na tipke lijevo, desno mijenja se trenutni model između čajnika i kvadrata (koji teksturu ima preslikanu 2x2 puta kako bi se vidjelo nastavljanje).

Izlaz iz programa je tipkom escape, a pritiskom na tipku "r" pokreće se i zaustavlja rotacija objekta.

Program se mora pokrenuti u direktoriju u kojem se nalaze i dvije prateće datoteke: *kernels.cl* i *tex1.txs*. U prvoj se nalazi OpenCL kod za generiranje tekstura a drugoj datoteci se nalaze upute za generiranje teksture.

Upute se navode jedna po retku datoteke, format je:

instrukcija parametri

Ako linija započinje znakom #, smatra se komentarom i preskače, a preskaču se i prazne linije.

Evo popisa dostupnih komandi i pripadnih parametara (parametri su opisani u pripadnim poglavljima za dani algoritam):

INST.	PARAMETRI	OPIS
create	ime_teksture	Stvara teksturu *ime_teksture*.
noise	tex, octaves, zoom, (boja1), (boja2)	Generira Turbulence šum.
cell	tex, mode, density, cellSize, (boja1), (boja2)	Mode označava način izračuna i može biti: - min_distance // udaljenost do najbliže točke - distance_diff // min2 - min1 - distance_sum // min1 + min2 - distance_product // min1 * min2 - second_min_distance // min2
gradient	tex, numPoints, { point1 (boja1) }...	stvara gradijent iz zadanih *numPoints* točaka, te zatim pridjeljuje boje teksturi prema opisanom algoritmu
intensity	tex, intensity	Mijenja intenzitet teksture

INST.	PARAMETRI	OPIS
distort	tex1, tex2, xpower, ypower	Distorzira teksturu *tex1* pomoću teksture *tex2*.
sine	tex, powerx, powery	Generira sinusni uzorak prema opisanom algoritmu koristeći zadane parametre.
blend_multiply	tex1, tex2	Multiplikativno miješa teksture *tex1* i *tex2*.

Dok se program izvodi, pritiskom na tipku "c" ponovno se parsira datoteka tex1.txs te se sve naredbe ponovno izvode (prije toga se uništavaju sve stare teksture).

Uz program dolaze i dvije projekt datoteke: jedna za Xcode / Mac OS X i jedna za Visual Studio / Windows.

Pod operacijskim sustavom Windows, potrebno je dodatno instalirati programski paket GLUT te OpenCL SDK za korištenu grafičku karticu.

5. MJERENJE I REZULTATI

Pri uspoređivanju brzine izvođenja algoritama u C i OpenCL implementaciji, htio sam prikazati kolika je razlika za pojedini algoritam, a zatim i za nekoliko kompozicija koje predstavljaju konačne teksture. Uz to, izmjerio sam brzine izvođenja na dva različita računala i na dva operacijska sustava. Treba imati na umu da ovdje na brzinu utječu vanjski čimbenici kao što su prevoditelji (i nativnog i OpenCL koda) te upravljački programi. To su komponente koje se mijenjaju i brzo razvijaju; za očekivati je da će s vremenom prevoditelji generirati brži kod a upravljački programi znati bolje odabrati način ostvarenja pojedinih zahtjeva upućenih od strane OpenCL-a odnosno programera. Još brže od razvoja programske podrške, razvija se i novo sklopovlje koje još više utječe na sve veću brzinu izvođenja. To se veoma dobro vidi i u prikazanim rezultatima gdje su brzine bitno različite, pri čemu je jedno računalo staro godinu dana a drugo četiri godine.

Konfiguracije računala i algoritmi

Računala

iMac

Intel Core i7 Extreme, Quad Core 2.93GHz + Hyperthreading

ATI HD 5750, 1GB GDDR3

4GB DD3 RAM, 1333MHz

MacBook Pro

Intel Core 2 Duo, Dual Core 2.4Ghz

nVidia 8600GT M, 256MB RAM GDDR2

2GB DD2 RAM, 667MHz

Operacijski sustavi

Mac OS X Lion, 10.7.1

Windows 7 64bit, najnovija verzija ATI Catalyst 11.8

Na računalu iMac sam mjerenja vršio na oba operacijska sustava, a na MacBook Pro-u samo na Lion-u. Radi čitljivosti, u grafovima sam kombinacije računala i operacijskih sustava označio ovako:

iMac + Windows7: iMac/Win

iMac + Lion: iMac/Lion

MacBook Pro + Lion: MBP/Lion

Na ove tri konfiguracije, mjerio sam brzine generiranja tekstura u šest različitih datoteka. Sve generirane teksture su veličine 1024x1024 slikovnih elemenata.

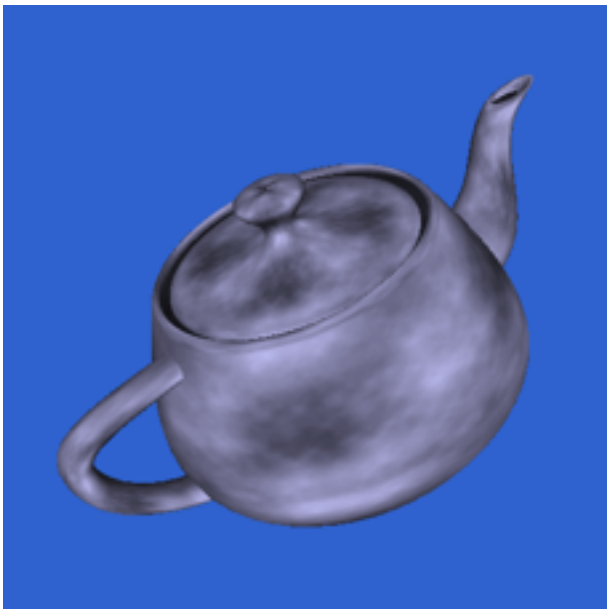
Pod Windows operacijskim sustavom korišten je Microsoft Visual Studio 2008, a opcije prevoditelja postavljene na standardne u "Release" konfiguraciji.

Pod Mac OS X Lion-om korišten je Xcode 4.2, prevoditelj Apple LLVM 3.0. Postavke također standardne za "Release" konfiguraciju.

Svi rezultati su u sekundama.

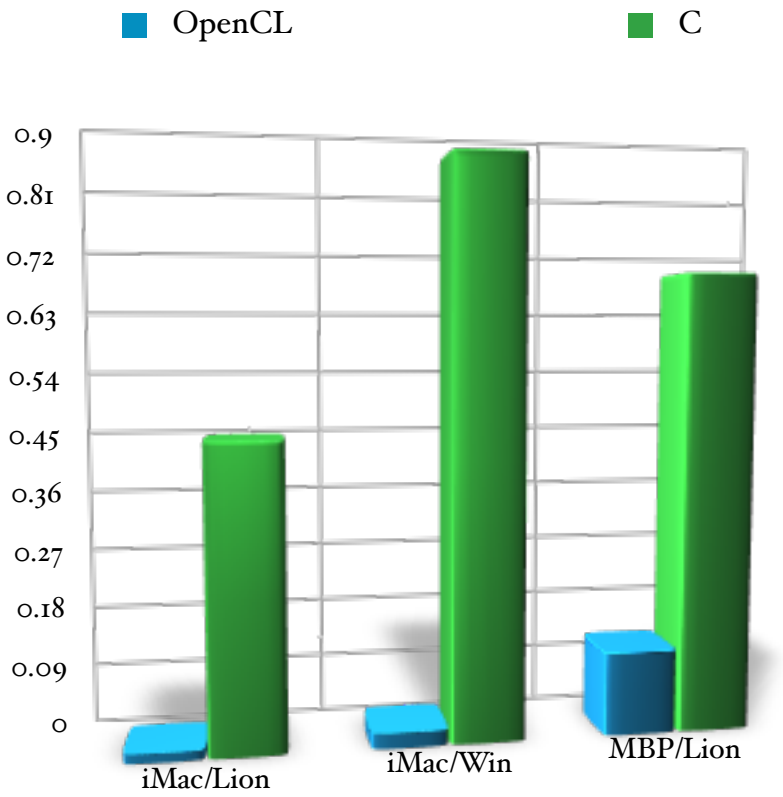
Konfiguracija 1

Samo Turbulence algoritam. Koristio sam 16 oktava a početni faktor povećanja je 0.5.



Slika 13. Konfiguracija 1

```
1024

## turbulence

create tex1

noise tex1 16 0.5 (0, 0, 00) (200, 200, 255)

output tex1
```



Slika 14. Rezultati mjerenja za Konfiguraciju 1

	iMac/Lion	iMac/Win	MBP/Lion
OpenCL	0.01417	0.02331	0.12618
C	0.45115	0.85821	0.68884

Vidimo ogromne razlike u brzini između OpenCL i C implementacije, osim na najslabijem računalu kod kojega je grafička kartica nekoliko generacija starija i još k tome u izvedbi za prijenosna računala, pa ima manje paralelnih razina ostvarenih sklopovski.

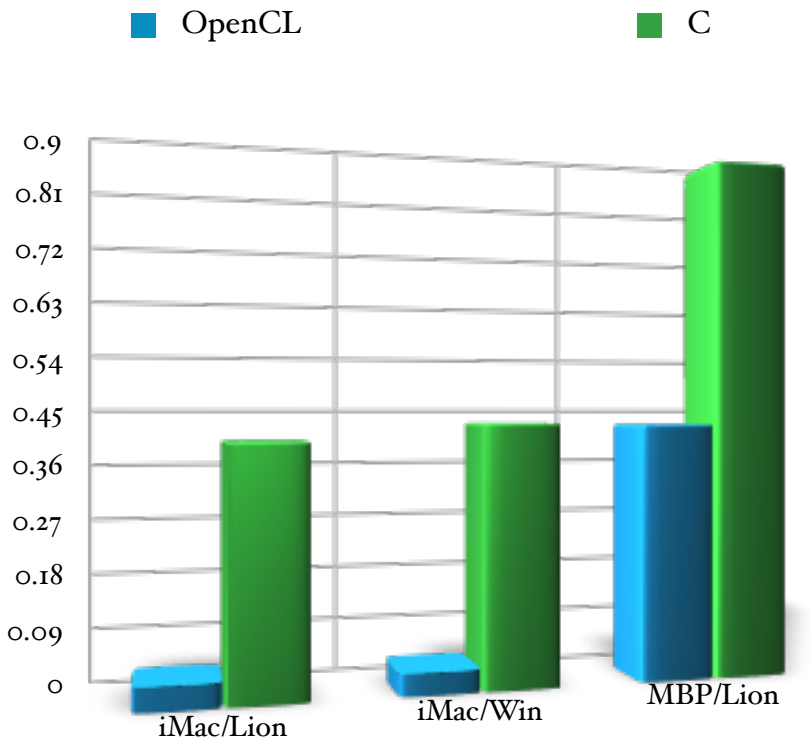
Iako je i centralna upravljačka jedinica starija nekoliko generacija od one u iMac-u, C implementacija je jednodretvena, pa razlika u brzini nije toliko vidljiva. Dio razloga je i u tome što se grafičke kartice trenutno razvijaju znatno brže od centralnih upravljačkih jedinica.

Konfiguracija 2

Samo Worely algoritam. Koristio sam veličinu ćelija 32 slikovna elementa i gustoće 0.4.

Slika 15. Konfiguracija 2

```
1024

## cell

create tex1

cell tex1 min_distance 0.4 32 (30, 0, 0) (170, 90, 0)

output tex1
```



Slika 16. Rezultati mjerenja za Konfiguraciju 2

	iMac/Lion	iMac/Win	MBP/Lion
OpenCL	0.03952	0.03716	0.42786
C	0.40318	0.43035	0.87390

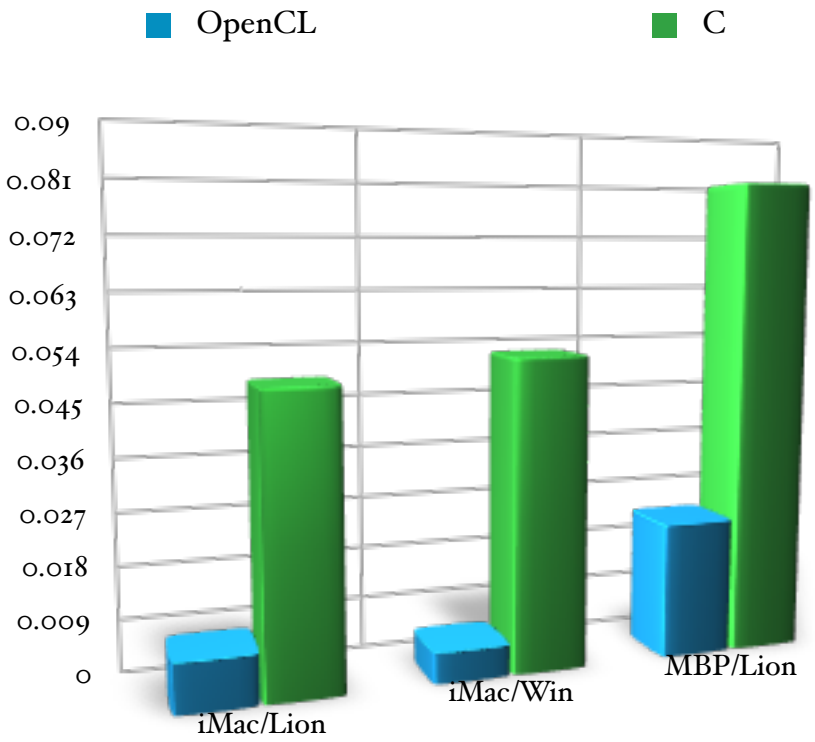
Rezultati su slični kao i kod prethodne konfiguracije, osim što pod Windows operacijskim sustavom vidimo primjetno bolje performanse u C implementaciji. Teško je odrediti što je tome uzrok, no vjerojatno je prevoditelj uspio bolje optimizirati kod.

Konfiguracija 3

Samo sinusni uzorak. Broj perioda u x i y smjeru je 8, pa dobijamo uzorak pod kutom od 45 stupnjeva.

Slika 17. Konfiguracija 3

```
1024

## sine

create tex1

sine tex1 8 8

output tex1
```



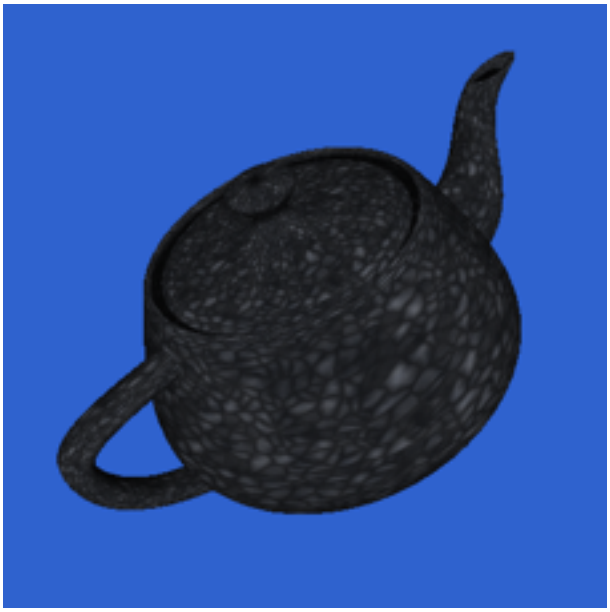
Slika 18. Rezultati mjerenja za Konfiguraciju 3

	iMac/Lion	iMac/Win	MBP/Lion
OpenCL	0.00829	0.00521	0.02325
C	0.04924	0.05263	0.08103

Očekivano, ovo je najbrži generacijski algoritam jer je i izračun najjednostavniji, a sastoji se od samo nekoliko množenja i izračunavanja funkcije sinus. Odnosi su slični kao i u prethodnim primjerima.

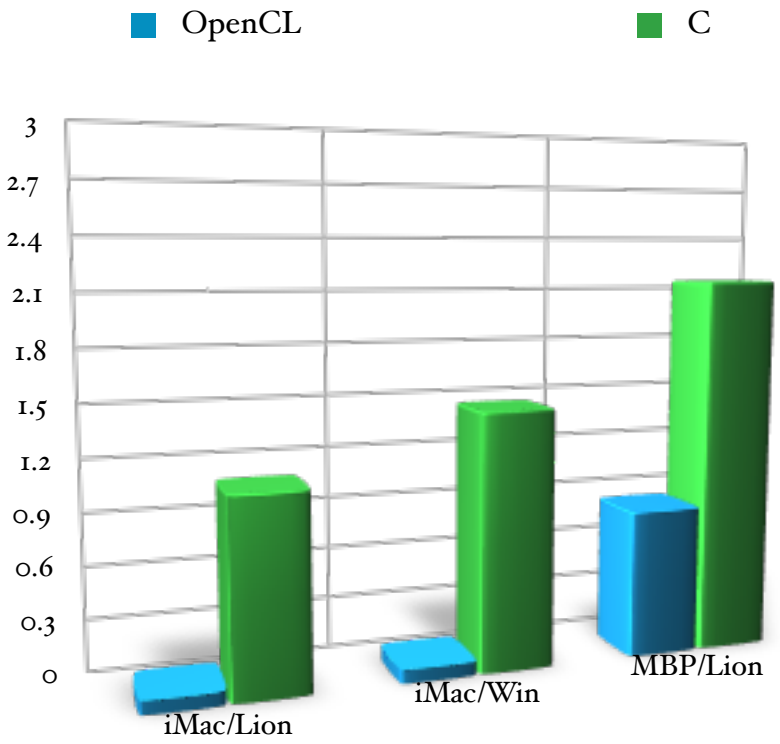
Konfiguracija 4

Tekstura koja podsjeća na kamenje. Kao osnova se koristi Worley algoritam, zatim se generira tekstura koristeći Turbulence algoritam i njome se distorzira originalna tekstura kako bismo dobili prirodniji oblik kamenja. Nakon toga kreiramo još jednu teksturu koristeći Turbulence algoritam ali ovaj puta s više sitnog šuma te ju multiplikativno pomiješamo s teksturom dobijenom u prethodnom koraku. Ovime dobijamo hrapavu površinu kamenja.



Slika 19. Konfiguracija 4

```
1024

## stones

# the texture for distorting the stones

create tex1

noise tex1 6 0.25 (0, 0, 0) (255, 255, 255)

# basic stones

create tex5

cell tex5 distance_diff 0.9 128 (40, 40, 40) (255, 255, 255)

distort tex5 tex1 0.4 0.4

create tex6

noise tex6 4 0.0675 (128, 138, 158) (235, 240, 255)

blend_multiply tex5 tex6

output tex5
```



Slika 20. Rezultati mjerenja za Konfiguraciju 4

	iMac/Lion	iMac/Win	MBP/Lion
OpenCL	0.07634	0.07795	0.84616
C	1.09930	1.46127	2.14751

Kako se radi o kompoziciji odnosno izvođenju nekoliko algoritama za redom, brzine generacije su očekivano manje nego u prethodnim primitivnim primjerima. Možemo primijetiti još veći raskorak između OpenCL i C implementacije, što možemo pripisati većoj mogućnosti paralelizacije izvođenja algoritma na grafičkom sklopovlju, odnosno situaciji koja pogoduje njegovom potpunijem iskorištenju. U suprotnosti s tim, situacija se ne mijenja značajno na najslabijem računalu - u ovom slučaju smo vjerojatno već premašili sklopovske mogućnosti grafičke kartice za daljnjom paralelizacijom izvođenja.

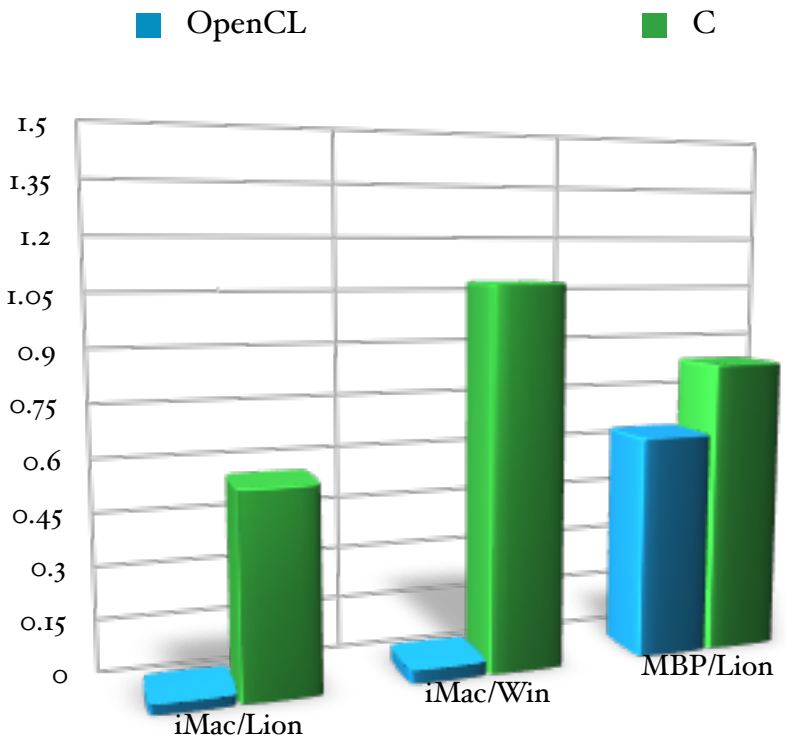
Konfiguracija 5

Tekstura koja podsjeća na svilenkastu tkaninu s tigrastim uzorkom. Kao osnova služi sinusni uzorak koji na sličan način kao u prethodnoj konfiguraciji distorziramo koristeći Turbulence teksturu. Time dobijamo prirodniji, nepravilniji izgled tigrastih šara. Zatim teksturu preslika-

vamo koristeći gradijent boja i naposljetku je multiplikativno miješamo s još jednom Turbulence teksturom, kako bismo dobili fine detalje na površini i dojam savijanja tkanine.

Slika 21. Konfiguracija 5

```
1024

## fabric

# the texture used for sine distortion

create tex1

noise tex1 6 0.5 (0, 0, 0) (255, 255, 255)

# finer noise

create tex4

noise tex4 4 0.0675 (98, 98, 98) (255, 255, 255)

# the base sine

create tex3

sine tex3 8 8

distort tex3 tex1 0.25 0.25

gradient tex3 5 {0 (80, 50, 0)} {40 (120, 60, 0)} {80 (150, 70, 0) } {120 (180, 100, 0) } {255 (200, 120, 0) }

blend_multiply tex3 tex4

output tex3
```



Slika 22. Rezultati mjerenja za Konfiguraciju 5

	iMac/Lion	iMac/Win	MBP/Lion
OpenCL	0.02920	0.03829	0.63727
C	0.56509	1.07600	0.83854

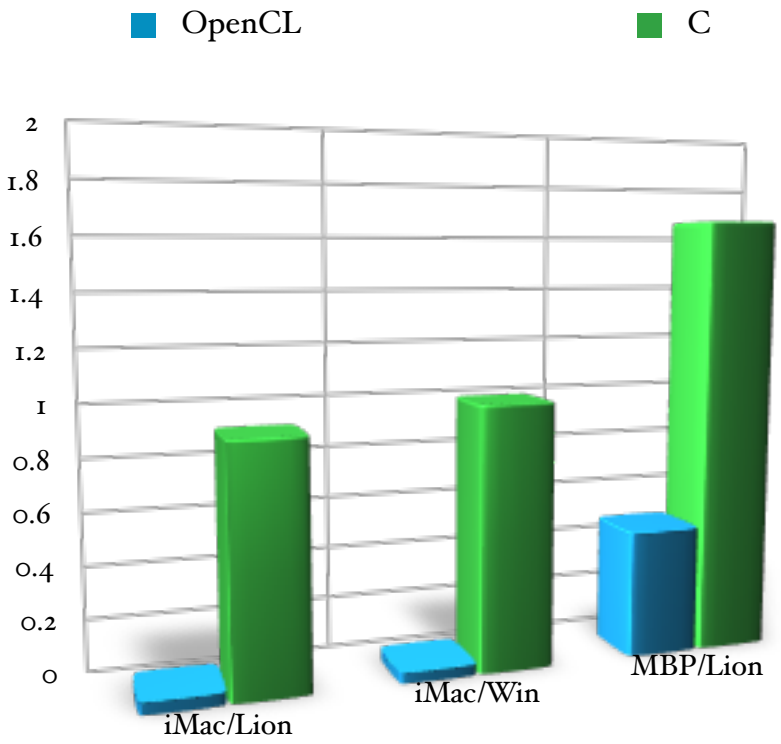
Slično kao u prošlom primjeru, osim što su ukupne brzine nešto veće jer se radi o jednostavnijim algoritmima. Iznimka je najslabija grafička kartica koja sve više zaostaje.

Konfiguracija 6

U posljednjoj konfiguraciji nastojao sam simulirati izgled kože (štavljene). Za osnovni uzorak sam uzeo Worley algoritam, a zatim sam dobivenu teksturu nekoliko puta množio i distorzirao Turbulence teksturama s različitim parametrima. Prvom sam dobio krupne prijelaze koji izgledaju poput blagog svijanja kože a drugom ponovno fine detalje na površini.

Slika 23. Konfiguracija 6

```
1024

#leather

create tex1

cell tex1 distance_diff 0.15 16 (140, 80, 00) (160, 105, 0)

create tex2

noise tex2 8 0.5 (208, 208, 208) (255, 255, 255)

blend_multiply tex1 tex2

noise tex2 3 0.0001 (198, 198, 198) (255, 255, 255)

distort tex1 tex2 0.1 0.1

blend_multiply tex1 tex2

output tex1
```



Slika 24. Rezultati mjerenja za Konfiguraciju 6

	iMac/Lion	iMac/Win	MBP/Lion
OpenCL	0.04601	0.04130	0.48500
C	0.91934	0.99800	1.65848

Nešto složenija kompozicija od prošle (u smislu složenosti izračuna), rezultati su lošiji ali su odnosi slični.

6. ZAKLJUČAK

(proceduralno generiranje tekstura)

Područje generacije proceduralnih tekstura je veoma široko i razvoj cjelokupnog seta alata i programskih knjižnica dostatnih za produkcijsku upotrebu je višegodišnji projekt za cijeli tim ljudi. Ovdje sam nastojao pružiti kratki uvod u područje kroz nekoliko različitih algoritma koji služe kao osnova za većinu primjena, a uz program priložen uz rad nalazi se nekoliko primjera tekstura koje se mogu na ovaj način generirati. Konfiguracijske datoteke olakšavaju isprobavanje načina rada dostupnih algoritama i mogućnosti njihovog kombiniranja.

Također, izmjerio sam brzine izvođenja ovih algoritama i ustvrdio da se pomoću OpenCL-a i modernog grafičkog sklopovlja teksture mogu generirati nekoliko desetaka puta brže - ovisno o dostupnom sklopovlju - donoseći cijeli proces u područje primjene u realnom vremenu. Htio bih naglasiti da se sasvim sigurno i C++ i OpenCL implementacije mogu dodatno optimizirati i time postići još veća brzina. Međutim, premda bi optimizacija bila obvezan korak u produkcijskom okruženju, u rezultatima rada nije osobito važno jer ne mijenja općeniti odnos brzine između ove dvije implementacije. Isto tako, dostupno sklopovlje se mijenja velikom brzinom i napreduje, pa će ovi rezultati isto tako brzo zastariti, no za očekivati je da će barem još neko vrijeme grafičko sklopovlje biti višestruko brže od centralnog procesora pri generiranju proceduralnih tekstura i sličnim algoritmima (na ovo upućuju i odnosi brzina OpenCL i C implementacija na novijem, odnosno starijem računalu), a OpenCL i/ili slični jezici će ući u puno širu upotrebu nego danas kako bi se ubrzali specifični zadaci koji se oslanjaju na veliku količinu proračuna.

Ključne riječi: proceduralne teksture, OpenCL, OpenGL, usporedba brzine, Turbulence, Worley

7. ABSTRACT

(procedural texture generation)

The subject of procedural texture generation is a bih one and the development of the whole toolset and programming libraries adequate for production use would be a multi-year project for a team of developers. Here I have tried to give a short introduction to the field through several different algorithms that are typically used as a base for most uses, and the application code attached to the paper contains several examples of textures generated this way. Configuration files make it easier to try out the implemented algorithms and the way they interact to produce the final output.

Additionally, I have measured the execution time of the algorithms and concluded that an OpenCL implementation running on modern graphics hardware can generate the textures several dozen times faster than the standard C implementation running on the CPU (this obviously depends on both the CPU and GPU hardware the algorithms are running on). I'd like to emphasize that both the C/C++ and OpenCL implementations could be optimized to achieve greater speed; however, although such optimizations would be a mandatory step in a production environment, they aren't relevant for this discussion because they don't significantly change the test results. Also, the hardware available is changing very quickly so the numbers presented here are bound to get out of date just as fast, but it's nonetheless very likely that the GPU implementation is going to remain several times faster than the CPU one for at least several more years. This is also hinted at by the numbers measured on older, compared to the numbers on the newer hardware. As a result, I expect the OpenCL and/or similar languages to experience a much higher adoption level than today as they will be used for accelerating specific tasks that are computationally intensive.

Keywords: procedural textures, OpenCL, OpenGL, speed comparison, turbulence, worley

7. LITERATURA

1. Texturing and Modeling, Third Edition: A Procedural Approach (David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steve Worley)

2. OpenCL Programming Guide (Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, Dan Ginsburg)

3. http://developer.nvidia.com/opencl

4. http://www.amd.com/us/products/technologies/stream-technology/opencl/Pages/opencl-intro.aspx

5. http://lodev.org/cgtutor/randomnoise.html

6. http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

7. http://www.doc.ic.ac.uk/~dt10/research/rngs-gpu-mwc64x.html

8. http://www.khronos.org/registry/cl/

9. http://www.thebigblob.com/getting-started-with-opencl-and-gpu-computing/

10. http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/

11. http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf

12. http://developer.apple.com/library/mac/documentation/Performance/Conceptual/OpenCL_MacProgGuide/OpenCL_MacProgGuide.pdf

13. http://devmag.org.za/2009/04/25/perlin-noise/