SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 564

SLOBODNO PROSTORNO MODELIRANJE

Mario Volarević

Zagreb, lipanj 2013.

ZAHVALA

Prvenstveno zahvaljujem mentorici prof. dr. sc. Željki Mihajlović na idejama i smjernicama prilikom izrade diplomskog rada te na pruženom znanju i stručnoj pomoći kroz sve ove godine studiranja.

Zatim zahvaljujem obitelji na ljubavi, financijskoj potpori i stvaranju sigurne okoline kojom su mi omogućili studiranje bez stresa i dali mi mogućnost da se usredotočim na ono što me zanima.

I naposljetku zahvaljujem prijateljima koji su mi svojim društvom olakšali studij i učinili da sve ove godine brzo prođu.

Sadržaj

1. Uvod

Iako smo danas svakodnevno okruženi računalima, a time i u neprestanoj interakciji s njima te su izmišljeni različiti načini upravljanja koji bi je olakšali i učinili intuitivnijom (kao što su miš, tipkovnica, igraća palica, igraći kontroler, „*3D* miš“ ...), još uvijek nemamo osjećaj direktne kontrole kao što je slučaj s manipulacijom objektima u stvarnom svijetu.

No u posljednjih nekoliko godina uvelike je povećana rasprostranjenost pametnih telefona i tableta te na taj način sve više dolazimo u kontakt s ekranima na dodir koji donekle poboljšavaju osjećaj kontrole jer se pokreti prstiju mogu direktno preslikati u odgovarajuće translacije na uređaju, ali nažalost, samo u dvije dimenzije. Ali zato ubrzo nakon toga dolaze i metode kontrole u sve tri dimenzije, kao što je slučaj s nekim pametnim telefonima, kontrolerom za *Nintendo Wii* i slično, koji koriste tehnologiju žiroskopa i akcelerometara za određivanje položaja u prostoru. No problem s takvim tehnikama je što ruke nisu potpuno slobodne, a kontrola s više udova istovremeno je nepraktična ili čak nemoguća.

Tada se na sceni pojavljuje *Microsoft* i predstavlja *Kinect* koji je najavljen kao uređaj za kontrolu pomoću cijelog tijela kojim je bilo moguće u virtualnom svijetu izvesti sve što radimo u stvarnom svijetu čime je napokon stvorena mogućnost intuitivne i izravne prostorne manipulacije u virtualnim okruženjima što je naposljetku omogućilo ostvarivanje slobodnog prostornog modeliranja.

Kako oko izvedbe modeliranja ima mnogo posla radi boljeg razumijevanja rad je strukturiran na način da će u prvom dijelu biti opisane teorijske osnove za svaku od tehnika koje su bile korištene da bi se modeliranje moglo izvesti, a u drugom dijelu će biti detalji tehničke implementacije za svaku od navedenih tehnika te problemi koji su se pojavljivali prilikom konkretnog razvoja.

2. Slobodno prostorno modeliranje

Ideja slobodnog prostornog modeliranja je da se modeli ne stvaraju dosadašnjom klasičnom računalnom metodom u nekom programu za *3D* modeliranje stvaranjem vrhova, krivulja i poligona i njihovom manipulacijom pomoću miša (preslikavanje iz *2D* prostora pomaka miša u položaje u *3D* prostoru scene uz konstantne manipulacije pogledom da se pokazivač pozicionira na pravo mjesto) nego se pomaci ruku u sve tri dimenzije u stvarnosti direktno preslikavaju u pomake virtualnih ruku na virtualnoj sceni te model oblikujemo dodavanjem i oduzimanjem smjese od koje je građen, slično kao što u bi u stvarnosti oblikovali glinu.

No, kao što je u uvodu već navedeno, da bi se to ostvarilo, prvo je bilo potrebno izumiti odgovarajuću ulaznu periferiju te je učiniti masovno dostupnom, relativno jeftinom i jednostavnom za korištenje, a jedan od glavnih predstavnika je *Microsoft Kinect* (ubrzo nakon njega su se pojavili i drugi slični uređaji).

Za ostvarivanje slobodnog prostornog modeliranja nije dovoljan samo odgovarajući „*hardware*“, nego je potrebno razviti i „*software*“. Glavninu čine algoritmi koje je moguće podijeliti u dvije veće kategorije. Jedna je vezana za ostvarivanje prihvatljivog ponašanje virtualne mase za modeliranje, a u drugu spadaju tehnike za detekciju prstiju ruku. A osim tih algoritama neki od bitnijih dijelova su implementacija gesti, vizualizacija dubinske mape i spremanje modela u format kompatibilan s klasičnim programima za *3D* modeliranje.

Kako je osnova za stvaranje ovog rada *Microsoft Kinect* potrebno je upoznavanje s načinom rada da bi kasnije bilo jasnije kako i zašto su neke stvari izvedene u programu.

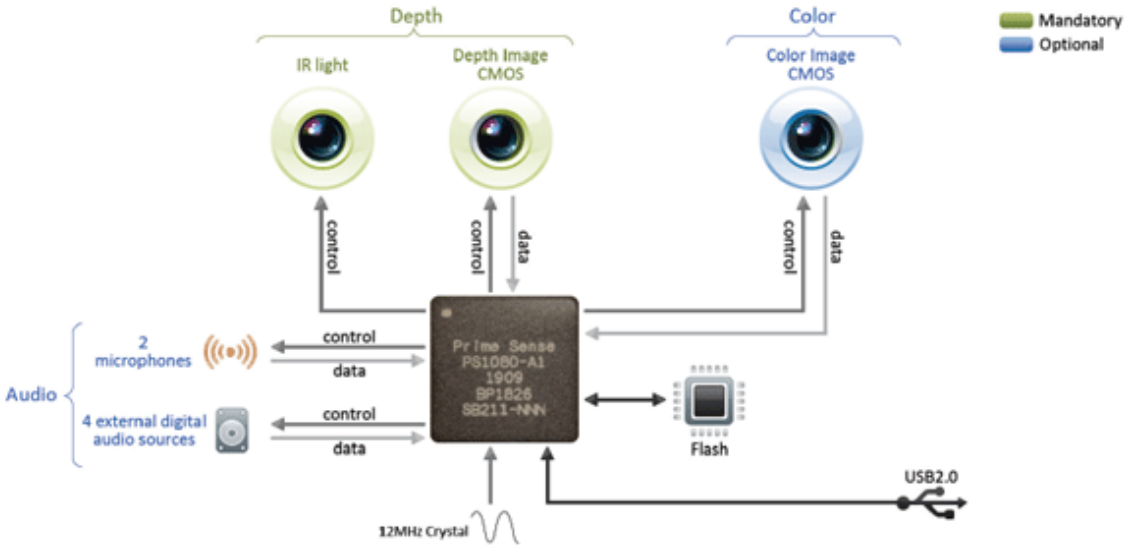
3. Microsoft Kinect

Glavna značajka *Kinecta* koja ga čini jedinstvenim i posebno zanimljivim za ovakvu vrstu rada jest ta što je uz klasičnu *2D* sliku moguće dobiti i informaciju o dubini, tj. trećoj

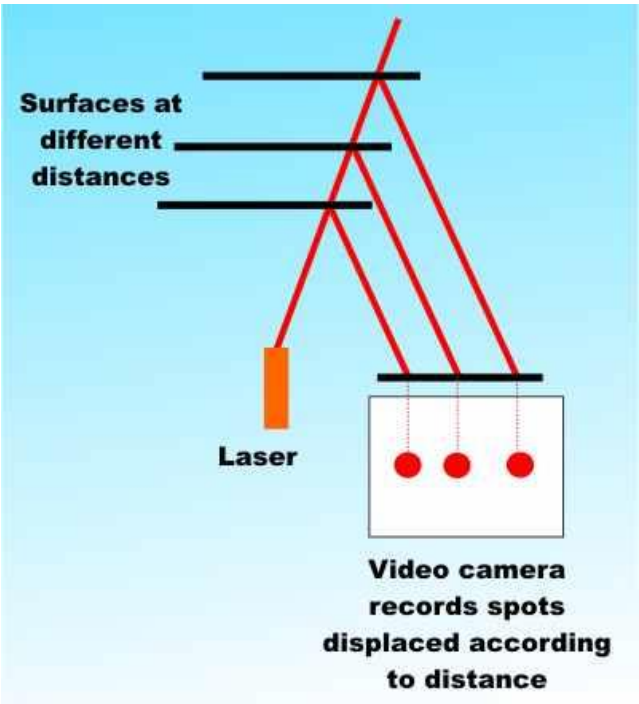
dimenziji, a uz to još i daje položaje glavnih zglobova korisnikovog tijela, a do svih tih podataka je lako pristupiti kroz *API* (engl. *Application Programming Interface*) te se na taj način mogu primijeniti u programu.

Slika 1 *Microsoft Kinect*

Tehnologija koja to omogućuje zasnovana je na inovativnom sklopovlju i algoritmima. Sklopovlje dubinske kamere je izvedeno pomoću laserskog infracrvenog izvora koji na scenu projicira slučajni uzorak točaka i infracrvenog detektora koji snima taj uzorak, ali s pomaknutim točkama zbog različite udaljenosti od izvora, te je iz tih pomaka moguće dobiti podatak o dubini za svaki slikovni element (engl. *pixel*) kamere (Slika 2, Slika 3).



Slika 2 Sklopovlje *Kinecta*



Slika 3 Određivanje dubine

Podatak o dubini je od iznimne važnosti jer olakšava posao algoritmima za segmentaciju scene i prepoznavanje objekata jer nije potreban računalno zahtjevan dodatan korak algoritama za detekciju rubova da se dobiju konture i izdvajanje od pozadine, a i rezultati su puno precizniji.

Praćenje kostura zasnovano je na brzom i učinkovitom klasifikatoru naziva *slučajne šume odlučivanja* (engl. *Randomized Decision Forests*). Svaka šuma se sastoji od više različitih stabala gdje se u listovima za svaki testirani element dolazi do zaključka kojem dijelu tijela pripada, a kao parametar za odlučivanje prelazaka iz grane u granu koristi se jednostavna formula koja uspoređuje odnose dubina između 3 lagano razmaknuta slikovna elementa koji sadržavaju i podatak o dubini (tim odnosima je moguće prepoznati tanke linije, koje šuma može prepoznati kao ruke, veće kontinuirane blokove, koji mogu predstavljati tijelo, itd.). No, veliki problem je treniranje takve šume da bi bila učinkovita za komercijalne primjene. „Treniranje 3 stabla do dubine 20 s milijun slika traje oko jedan dan na grozdu s 1000 jezgri“ [2].

Ali jednom kad je klasifikator istreniran, prepoznavanje je iznimno brzo i zbog svoje paralelne prirode se učinkovito izvršava na grafičkom procesoru (engl. *GPU*) gdje se svi slikovni elementi (engl. p*ixel*) obrađuju odjednom brzinom od 200 *FPS*-a (engl. Frame Per Second), što iznosi oko 5 ms po slici (engl. *frame*) na grafičkoj kartici u rangu *ATI Radeon HD 2900 GT* [2].

4. Masa za modeliranje

Važan kriterij kod odabira algoritma za prikaz i izradu mase bila je jednostavnost modifikacije modela uz laganu mogućnost stvaranja rupa ili dodavanje nove količine mase. Implementacija gdje bi se izravno mijenjala poligonalna mreža (engl. *Polygonal mesh*) bila bi u nekim slučajevima poprilično komplicirana, te je stoga odlučeno da će se koristiti volumni model koji bi se zatim naknadno poligonizirao.

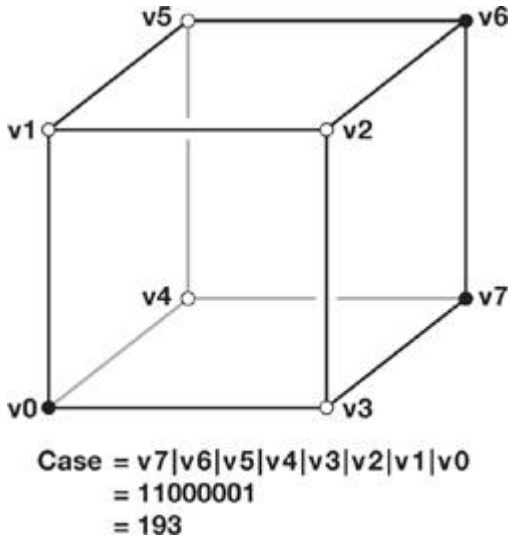
Volumni model je uglavnom predstavljen skalarnim vrijednostima u nizu dvodimenzionalnih kriški (engl. *slices*) jer se na taj način podaci dobivaju iz skenirajućih uređaja, ali mogu biti protumačeni i kao trodimenzionalno polje skalara. Skalari se mogu tumačiti kao razne vrijednosti, ovisno o namjeni, ali najčešće su to gustoće različitih tkiva. Također, jedna od čestih pretpostavki je da negativne vrijednosti predstavljaju prazna, a pozitivne ispunjena područja.

Postoji mnoštvo algoritama za vizualizaciju volumnih modela, a jedan od najpoznatijih i najjednostavnijih za poligonizaciju volumena je algoritam pokretnih kocki (engl. *Marching Cubes*). U usporedbi s nekim drugim tehnikama vizualni rezultati nisu toliko dobri, ali zato iznimno brzo radi, čak i u osnovnoj verziji bez dodatnih optimizacija. Kako je u ovom slučaju važnije bilo ostvariti brzinu radi mogućnosti interaktivnog prikazivanja scene nego visoku kvalitetu slike, odabran je ovaj algoritam.

4.1. Algoritam Marching cubes

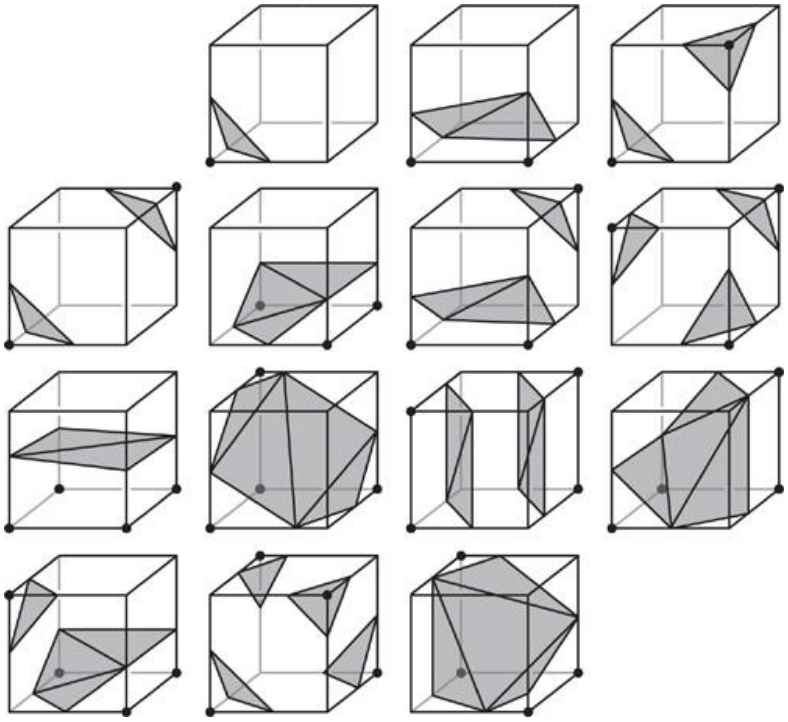
Algoritam je predstavljen 1987. g. kao jednostavno rješenje za vizualizaciju volumnih podataka, primarno medicinskih *3D* podataka prikupljenih iz tomografa ili magnetske rezonancije gdje se kao rezultat dobiva izo-površina (engl. *Isosurface*) predstavljena poligonalnom mrežom [7].

Algoritam se temelji na podjeli volumena na manje kockaste ćelije u kojima će se iscrtavati poligoni i usporedbi svake od tih ćelija s predefiniranom tablicom koja određuje kako će poligoni biti orijentirani, tj. na kojim bridovima kocke se nalaze vrhovi poligona. Uvjet za određivanje koje vrijednosti iz tablice koristiti određuje se tako da u vrhovima tih ćelija uzorkujemo vrijednosti *3D* volumnog polja te stvaramo *8-bitni* binarni broj (za svaki vrh kocke jedan bit) gdje je bit postavljen kao *1* ako je uzorkovana vrijednost u vrhu veća od neke prethodno zadane proizvoljne izo-vrijednosti, a na *0* ako je manja (Slika 4). Taj broj se zatim koristi kao indeks za tu tablicu. Potrebno je pripaziti kojim redoslijedom se označavaju vrhovi kocke jer je bitna konzistencija da se izabere točan indeks unutar tablice. U protivnom se dobivaju krivi poligoni na krivim mjestima.

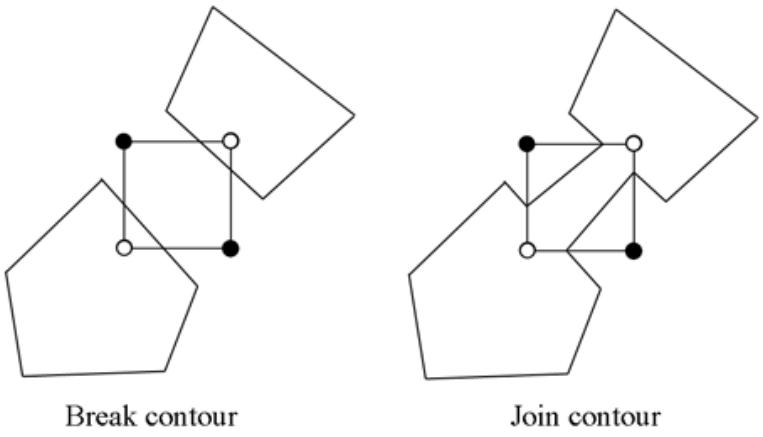


Slika 4 Primjer stvaranja indeksa

Iako postoji 2^8 različitih kombinacija sve se svodi na 15 osnovnih konfiguracija (jedna od njih je prazna kocka) koje se zatim mogu rotirati ili zrcaliti gdje se stvaraju do 4 poligona po kocki (Slika 5). No, prvobitni algoritam je imao greške zbog dvosmislenosti u nekim slučajevima (Slika 6) pa su neke konfiguracije popravljene te je u tim slučajevima moguće stvoriti do 5 poligona [16].



Slika 5 Prikaz mogućih konfiguracija poligona



Slika 6 Primjer dvosmislenog slučaja u *2D*

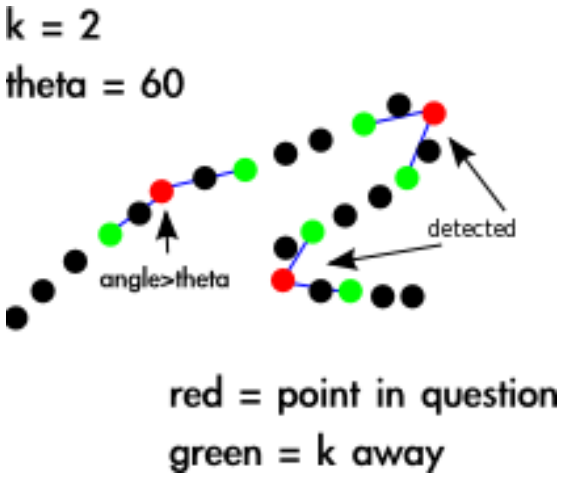
Osim prikaza geometrije, ovaj algoritam opisuje i postupak dobivanja normala vrhova (engl. *vertex normal*) koje se mogu koristiti za izračun osvjetljenja. Normale se dobiju na način da se normalizira gradijent koji se računa kao razlika vrijednosti volumena u susjednim točkama najbližima trenutno promatranom vrhu. Vrijednosti volumena se uzorkuju u sve *3* dimenzije i svaka od njih postaje *x*, *y* ili *z* komponenta gradijenta, a postupak se ponavlja za svaki vrh u sceni.

Kako su izračuni za svaku od ovih ćelija nezavisni, algoritam je vrlo lako paralelizirati te je pogodan za ubrzanje izvršavanja pomoću grafičkog procesora, što je i slučaj u ovom programu, o čemu će biti riječi kasnije.

4.2. Prepoznavanje prstiju i oblikovanje mase

Radi povećane kontrole i više mogućnosti prilikom oblikovanja modela odlučeno je da će program moći prepoznavati korisnikove prste. To je predstavljalo problem jer *Kinect* nema tu funkcionalnost ugrađenu te ju je potrebno ručno implementirati.

Prepoznavanje je izvedeno na način da se u kvadratu određene veličine opisanom oko točke ruke koju smo dobili od *Kinecta* napravi segmentacija po dubini te se izdvoji obris ruke, a zatim se na toj konturi izvede algoritam *k-zakrivljenosti* (engl. *k-curvature*) kojim se poprilično uspješno dobiju točke koje predstavljaju vrhove prstiju. Algoritam se temelji na usporedbi kuta kojeg zatvaraju *2* vektora određena s *3* točke na konturi razmaknuta za parametar *k.* Ti vektori se mogu zapisati u obliku $[C_i(j), C_i(j - k)]$ i $[C_i(j), C_i(j + k)]$, gdje vrijednosti u okruglim zagradama predstavljaju indekse u konturi koja je predstavljena listom (Slika 7).



Slika 7 Izvođenje postupka *k-zakrivljenosti*

Nakon što su dostupni podaci o *3D* položajima ruku i prstiju moguće ih je uz odgovarajuće skaliranje direktno koristiti kao indekse u volumnoj mapi za izravno mijenjanje vrijednosti čime se automatski mijenja i konačan vizualizirani model. Kako program predstavlja koncept, više pažnje je posvećeno mogućnostima nego funkcionalnoj upotrebi te je oblikovanje izvedeno na što jednostavniji način. Moguće je oblikovanje na nekoliko načina, lijevom rukom se dodaje masa, a desnom se oduzima kada dođe u kontakt s modelom. Kada je šaka stisnuta dodaju se ili oduzimaju veće količine na njihovim pozicijama, a kada su prsti ispruženi tada se koriste njihovi položaji za modeliranje s manjom količinom mase.

5. Dodatne značajke

Osim osnovne funkcionalnosti u program je uključeno i nekoliko dodatnih značajki koje ga čine upotrebljivijim i olakšavaju rad. Kao što je ranije spomenuto neke od važnijih su vizualizacija dubinske mape, geste za manipulaciju kamerom i položajem modela te spremanje poligonalnog modela sa scene u datoteku u formatu kompatibilnom s klasičnim programima za *3D* modeliranje.

5.1. Vizualizacija dubinske mape

Prikaz dubinske mape se aktivira samo kada se program pokrene na računalu na koje je spojen Kinect i koristi se za olakšavanje pozicioniranja osobe ispred kamere radi lakšeg početnog prepoznavanja ruku, a nakon što su ruke prepoznate. Na ovaj način znamo gdje su granice vidnog polja kamere. U prvoj fazi prozor prikazuje dubine u tonovima sive (engl. *greyscale*), a kasnije se pozadina boja crno, područja gdje je dubina nepoznata plavo, konture ruke žuto, a ruke se bojaju u jednu od 8 predefiniranih boja (radi potencijalne mogućnosti razlikovanja ruku više od jednog korisnika) (Slika 8).

Slika 8 Vizualizacija dubina

5.2. Manipulacija scene

Manipulacija scene se izvodi pomoću gesti s rukama i prstima ili pomoću tipki (*W, S, A, D* i *strelice*) na tipkovnici. Pomoću ruku se manipulira modelom na način da se s jednim prstom ispruženim, a jednom rukom sklopljenom rade rotacije, s dva prsta ispružena na odvojenim rukama koja se pomiču paralelno obavljaju translacije (u sve tri dimenzije), a ako se razmiču ili približavaju tada se skalira model.

Tipke na tipkovnici se koriste za manipulaciju kamerom. Slova translatiraju kameru, a strelice je rotiraju. Pomicanjem kamere je moguće gledati scenu iz raznih kutova, uključujući i položaj ruke ili prstiju, dok se gestama utječe samo na model.

5.3. Spremanje modela

Za spremanje modela odabran je *Wavefront .obj* format zbog svoje jednostavnosti, čitljivosti i lagane implementacije. Za potrebe ovog programa bilo je potrebno spremati samo najosnovniju geometriju. Znači, vrhove, normale i poligone. Format je tekstualan, a podaci se zapisuju u retke gdje svaki redak ima prefiks ovisno o tome što sadrži, npr. ***v*** predstavlja vrhove, ***vn*** predstavlja normale vrhova, a ***f*** predstavlja lica (engl. *faces*), odnosno poligone koji sadrže redni broj vrha, teksture i normale, što je vidljivo u primjeru:

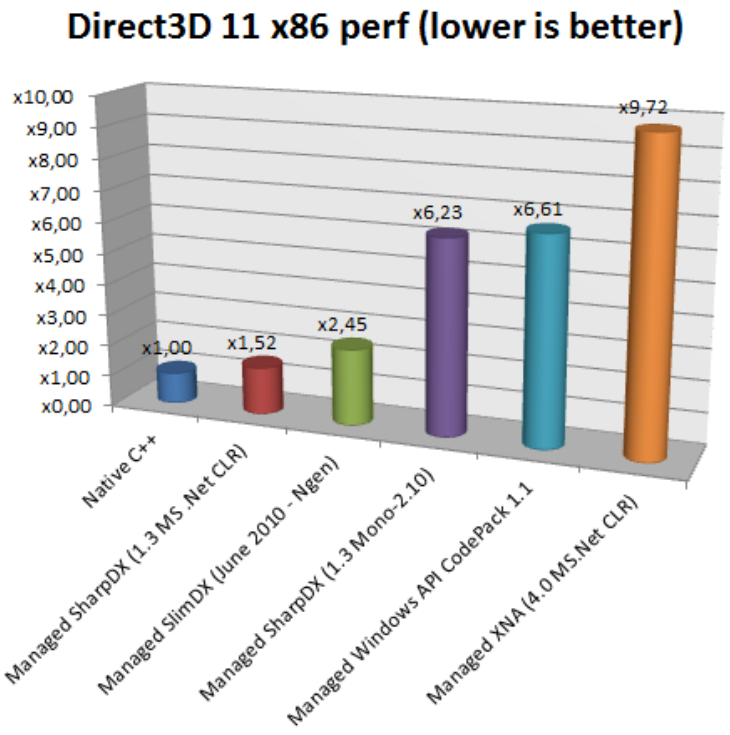
```
v 2.996907 2.996907 2.265347
vn -1 0 0
g Mesh
f 1//1 2//2 3//3
```

Postoji još nekoliko drugih prefiksa, ali oni u ovom slučaju nisu bili potrebni. Radi jednostavnosti izvedbe programa datoteka se sprema u isti direktorij gdje se nalazi program i nije moguća promjena naziva datoteke, ali zato svaki put prilikom spremanja generirano ime datoteke sadrži datum i vrijeme da bi bilo moguće spremati više različitih modela tijekom rada programa bez bojazni od prepisivanja (engl. *overwrite*).

6. Odabir tehnologija

Prije nego je moguće započeti s programiranjem treba odlučiti koje programske biblioteke (engl. *libraries*) i alate koristiti da bi se olakšalo programiranje uz zadržavanje performansi i kompatibilnosti. Kako je esencijalan dio rada podrška za *Microsoft Kinect*, odlučeno je da će biti korištene *Microsoftove* tehnologije radi olakšanog povezivanja i povećane kompatibilnosti. Stoga je za programski jezik odabran *C#* u *Visual Studio* razvojnom okruženju, korištena je verzija *2012* zbog odlične ugrađene podrške za ispravljanje grešaka (engl. *Debugger)* za programe za sjenčanje (engl. *shader)* što je iznimno bitno jer je programiranje grafičkog procesora poprilično nezgodan zadatak. Kao programsko sučelje (engl. *API, Application Programming Interface*) za povezivanje s *Kinectom* korišten je *Kinect SDK v1.7* jer je dobro dokumentiran, službeno podržava sve funkcije *Kinecta* te

ima bolje praćenje korisnika u odnosu na druge slične *API-je*. I naposljetku, kao grafički *API* je odabran *DirectX 10* uz korištenje programske biblioteke *SharpDX* jer je za rad trebala podrška za moderne mogućnosti grafičkih procesora (kao što je *Geometry shader*), a ujedno je bila poželjna što veća brzina (što je vidljivo u testovima, Slika 9) i lakoća korištenja. Osim grafičkog alata za ispravljanje pogrešaka (*debuggera*) ugrađenog u VS2012 korišten je i *AMD-ov* alat *GPU Perf Studio 2* jer u nekim slučajevima pruža više informacija i proširuje set mogućnosti što dovodi do boljeg razumijevanja grešaka i njihovog lakšeg i bržeg ispravljanja.



Slika 9 Usporedba brzina raznih *3D* biblioteka

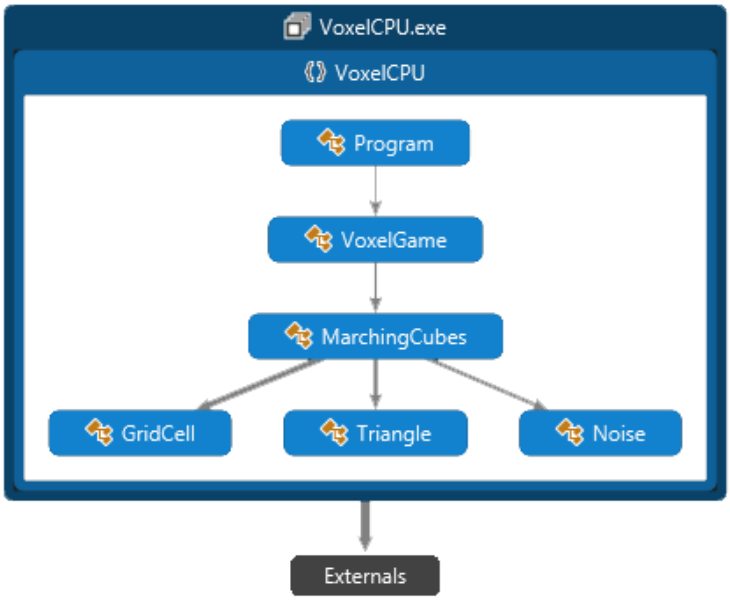
7. Programska implementacija

Tijekom izrade ovog rada napravljene su dvije verzije osnovnog programa s različitim izvedbama algoritma pokretnih kocaka *(Marching cubes)*. Jedna izvedba je bila implementirana na glavnom procesoru (engl. *CPU*) radi boljeg upoznavanja načina na koji algoritam radi, a druga i konačna (korištena u posljednjoj verziji programa) je prilagođena

za izvršavanje na grafičkom procesoru (engl. *GPU, Graphics Processing Unit*) kako bi se iskoristila ubrzanja koja pruža njegova paralelna arhitektura.

7.1. Algoritam *CPU* Pokretne kocke

Struktura ove verzije programa iznimno je jednostavna, kao što prikazuje Slika 10. U ovoj verziji samo je implementiran algoritam pokretnih kocki više-manje doslovno kako je opisan, uz nekoliko dodataka koji olakšavaju rad i poboljšavaju vizualni rezultat, kao što su mogućnost učitavanja volumnog modela iz datoteke ili stvaranje terena pomoću *Perlin simplex* algoritma za šum (engl. *Perling simplex noise*, značajka ovog algoritma je stvaranje niza vrijednosti u više smjerova bez pojavljivanja naglih skokova i diskontinuiteta), kontrola kamere i korištenje konstantnog sjenčanja (engl. *flat shading*).



Slika 10 Struktura klasa CPU verzije projekta

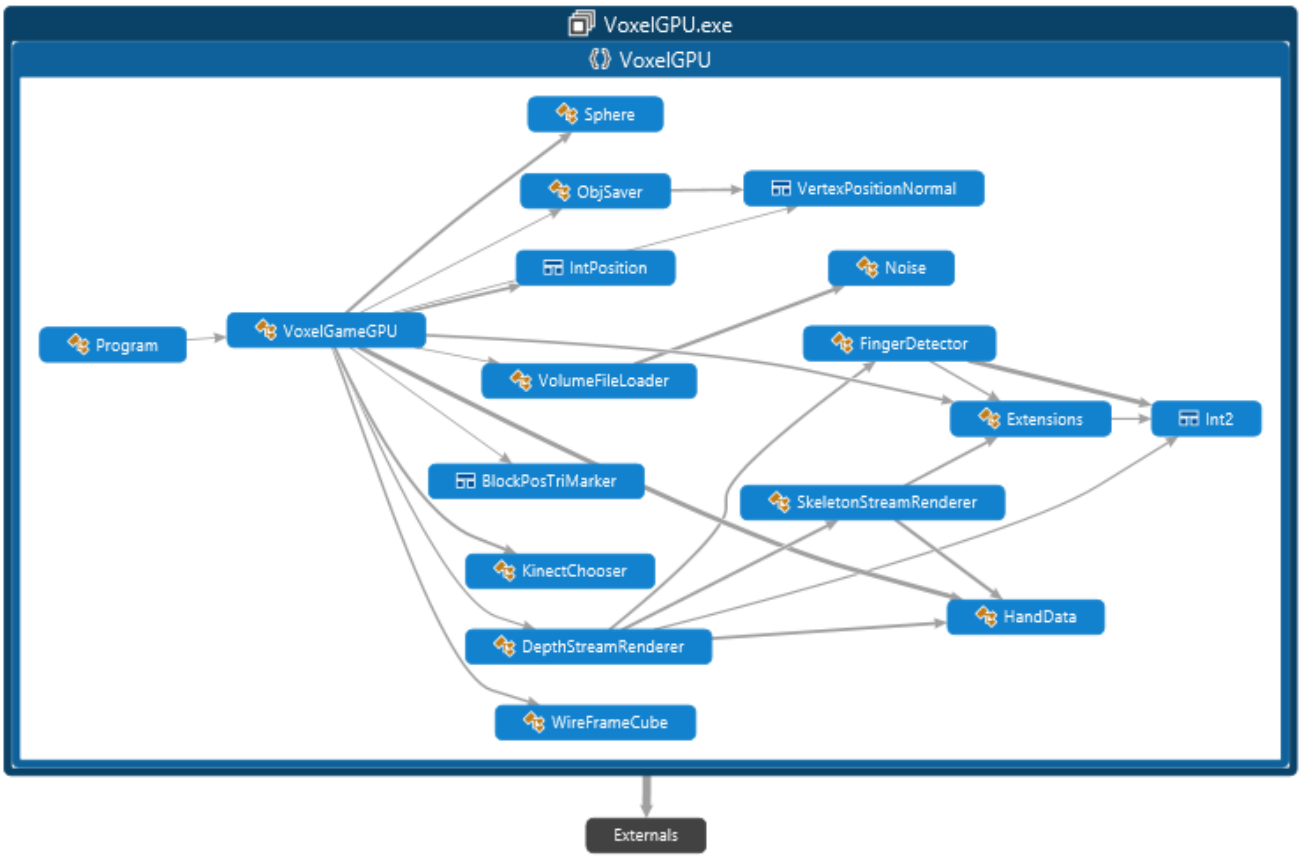
Problem je s ovom implementacijom što nije prilagođena za interaktivno korištenje. Podjela u ćelije, uzorkovanje volumnih podataka i stvaranje trokuta odvija se samo jednom, prilikom inicijalizacije programa, traje vrlo dugo (nekoliko sekundi) i troši puno radne memorije na računalu (engl. *RAM*), ali jednom kad se stvore spremnici vrhova (engl. vertex buffer) iscrtavanje modela je brzo i moguće su interaktivne manipulacije

pogledom, ali u slučaju da se nešto na modelu promijeni cijeli postupak dugotrajne početne inicijalizacije bi se morao ponoviti.

Iako su neka poboljšanja moguća, kao što je ponovno izračunavanje samo nekih ćelija, nije ih bilo smisla provoditi jer je ova verzija programa služila samo za testiranje funkcionalnosti.

7.2. Algoritam *GPU* Pokretne kocke

Struktura ove verzije programa nešto je kompliciranija jer je osim osnovnog algoritma implementirano i mnoštvo drugih mogućnosti koje su spomenute ranije u teorijskom dijelu (Slika 11). Također, veliki dio programa nije prikazan na dijagramu klasa jer je veliki dio algoritma izveden u programu za sjenčanje (engl. *shader program,* hrv. *sjenčar*).

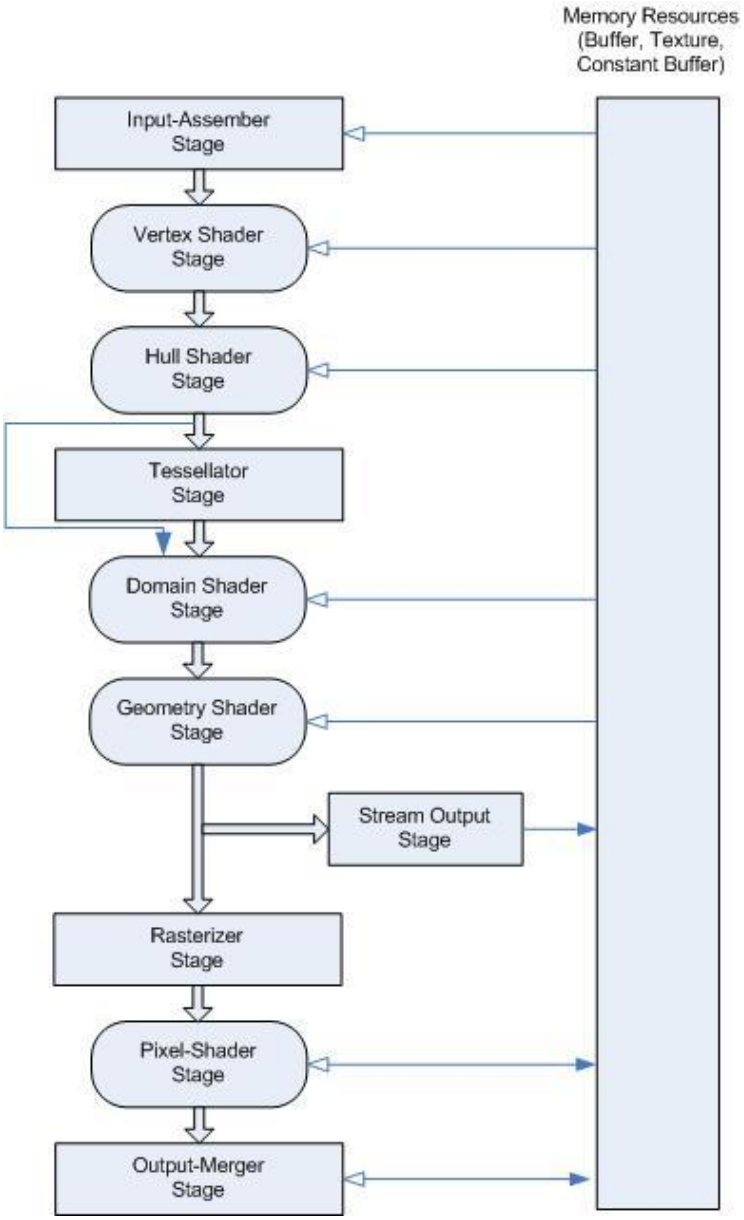


Slika 11 Struktura klasa GPU verzije projekta

7.2.1. Grafički protočni sustav

Programi za sjenčanje koriste se da bi se algoritam mogao izvršiti na grafičkom procesoru. Pomoću programa za sjenčanje moguće je programirati ponašanje raznih dijelova protočne strukture (engl. *pipeline*) grafičkog procesora (Slika 12). Prilikom iscrtavanja scene grafička kartica prolazi kroz navedene faze, s tim da je po potrebi neke moguće preskočiti. Također, potrebno je primijetiti da se naziv *shader* koristi i kao naziv programa i kao faze protočne strukture, odnosno fizičke grafičke jezgre. Razlog tome je staro nazivlje iz vremena kada su grafičke kartice imale specijalizirane aritmetičke jedinice samo za operacije s određenom vrstom elementa koji su bili direktno povezani s programom, a jedina zadaća im je bila kontrola sjenčanja, no moderne grafičke kartice više nemaju specijalizirane elemente samo za vrhove ili samo za slikovne elemente kao prije, nego se koristi ujedinjena arhitekura za sjenčanje (engl. *Unified Shading Architecture*), gdje jezgre mogu obavljati općenite operacije, slično glavnom procesoru, ali jednostavnije.

U ovoj implementaciji korišteni su programi za vrhove, geometriju i slikovne elemente (engl. *Vertex, Geometry i Pixel Shader*). *Vertex shader* koristi se za manipulaciju vrhovima, *geometry shader* može povećati ili smanjiti broj vrhova, a *pixel shader* služi za manipulaciju slikovnim elementima ekrana. Svaki program za sjenčanje zatim je moguće zasebno pozivati u programu, ali je moguće i stvoriti efekt koji predstavlja kompoziciju više programa za sjenčanje podijeljenih u prolaze (engl. *pass*) i zatim povezane u tehniku (engl. *technique*) radi lakšeg i urednijeg korištenja unutar programa.



Slika 12 Protočna struktura grafičkog procesora

Uz navedene programe za sjenčanje koristi se još i faza izlaza toka podataka (engl. *Stream Output Stage*). Kao što je vidljivo na slici, ta faza omogućuje vraćanje podataka u memoriju nakon što su obrađeni u programima za sjenčanje te zatim tim podacima može pristupiti glavni procesor, odnosno iskoristivi su u glavnom programu.

Glavna razlika u implementaciji ove verzije u odnosu na prethodnu koja se izvršava na glavnom procesoru je što se ovdje svi postupci algoritma izvršavaju u svakoj iteraciji što

kao posljedicu ima iznimno jednostavnu implementaciju izmjene geometrije modela, koja će kasnije biti opisana malo detaljnije. Kako je uspješna implementacija poligonizacije volumnog modela neophodna za uspješno funkcioniranje, a ujedno i jedna od osnovnih zadaća programa, prvo će biti opisana ta značajka, a zatim i ostale.

7.2.2. Poligonizacija volumnog modela

U implementaciji se prvo kreće s učitavanjem volumnog modela u trodimenzionalno polje iz datoteke ili stvaranja jednostavnog kockastog bloka proizvoljnih dimenzija. Zatim je učitani model potrebno prebaciti na grafičku karticu da bi programi za sjenčanje imali podatke s kojima mogu raditi. Radi toga se stvara *3D* tekstura te se šalje u program za sjenčanje, odnosno Efekt (ovisno što je u glavnom programu korišteno):

```
float[] texDat = VolumeFileLoader.CreateBlock(wx, wy, wz);
testTex = Texture3D.New<float>(GraphicsDevice, wx, wy, wz, PixelFormat.R32.Float,
     texDat, TextureFlags.ShaderResource, sdx11.ResourceUsage.Default);
voxelEffect.Parameters["densityVolume"].SetResource<Texture3D>(testTex);
```

Nakon slanja teksture i još nekih parametara stvaraju se spremnik vrhova i opis spremnika (engl. *layout*). Spremnik vrhova ima veličinu koja odgovara dimenzijama volumnog modela (iako je svaka od dimenzija umanjena za 2 radi kasnije mogućnosti uzimanja vrijednosti na rubnim lokacijama prilikom izračuna gradijenta za normale).

```
IntPosition[] voxPositions = new IntPosition[(wx - 2) * (wy - 2) * (wz - 2)];
int cnt = 0;
for (int x = 1; x < wx - 1; x++)
    for (int y = 1; y < wy - 1; y++)
        for (int z = 1; z < wz - 1; z++)
        {
            voxPositions[cnt++] = new IntPosition(new Int4(x, y, z, 0));
        }
vpcBuff = Buffer.Vertex.New(GraphicsDevice, voxPositions);
inputLayout = VertexInputLayout.FromBuffer(0, vpcBuff);
```

Vrijednosti u tom spremniku su analogne lokacijama ćelija koje će biti korištene u *MC* algoritmu te se koriste kao glavni ulazni parametar u program za sjenčanje vrhova. Time se omogućuje izvršavanje istog programa za sjenčanje za sve ćelije istovremeno.

Osim ovog spremnika još je potrebno inicijalizirati i spremnik vrhova za *Stream Output* fazu, a detaljniji razlog zašto će biti obrazložen kasnije:

```
soBuffer = Buffer.New(GraphicsDevice, wx * wy * wz * 2 * sizeof(uint),
 BufferFlags.StreamOutput | BufferFlags.VertexBuffer | BufferFlags.ShaderResource,
 PixelFormat.R32G32.UInt);
soInputLayout = VertexInputLayout.New<BlockPosTriMarker>(0);
```

Zadnji korak prije nego što započne računanje na grafičkoj kartici je slanje tih prethodno inicijaliziranih spremnika u grafičku memoriju i to se odvija u svakoj iteraciji petlje zadužene za crtanje:

```
GraphicsDevice.SetVertexBuffer(vpcBuff);
GraphicsDevice.SetVertexInputLayout(inputLayout);
GraphicsDevice.SetStreamOutputTarget(soBuffer, 0);
voxelEffect.CurrentTechnique.Passes[0].Apply();
((SharpDX.Direct3D11.DeviceContext)GraphicsDevice).PixelShader.Set(null);
GraphicsDevice.SetDepthStencilState(GraphicsDevice.DepthStencilStates.None);
GraphicsDevice.Draw(PrimitiveType.PointList, vpcBuff.ElementCount);
```

Iako je moguće jednostavno izvesti algoritam u jednom prolazu, brzina bi u tom slučaju bila iznimno loša. Naime, zbog korištenja *geometry shadera* brzina i učinkovitost jako ovise o razlici količine ulaznih i izlaznih *byte-ova* za svaki element [5][6]. A kako je za svaku ćeliju moguće stvoriti do 5 trokuta, znači da bi za svaki ulazni element bilo do 15 izlaznih elemenata od kojih bi svaki imao informaciju o položaju (vektor veličine 4 *float-a*) i normali (vektor veličine 3 *float-a*) što kombinirano iznosi 15*7=105 *float-ova* naspram 4 *int-a* (položaj ćelije i indeks konfiguracije u tablici), a takav nesrazmjer iznimno negativno utječe na performanse. Stoga je u *GPU* izvedbi bolje podijeliti algoritam u 2 prolaza, gdje će svaki od *geometry shadera* biti s manjom razlikom ulaznih i izlaznih parametara, čime bi u teoriji performanse trebale narasti čak 22 puta u odnosu na verziju s jednim prolazom [14].

Kod algoritma podijeljenog u 2 prolaza, prvi prolaz zadužen je za uzorkovanje podataka u ćelijama, izračun indeksa i stvaranje jednostavnih *markera* koji predstavljaju ćelije gdje će se stvarati poligoni, a drugi prolaz stvara vrhove, poligone, normale i iscrtava konačnu geometriju. Rast performansi proizlazi iz dva razloga. Prvi je što će broj markera koji izlaze

iz *geometry shadera* biti puno manji od ukupnog broja ćelija jer se sve potpuno prazne ili potpuno pune ćelije odbacuju (u varijanti s jednim prolazom kasniji koraci algoritma bi se izvršavali za sve ćelije). A drugi je razlog korištenje pakiranja parametara za ulazne i izlazne argumente *geometry shadera* da bi se smanjio njihov ukupan broj (sa svrhom povećanja učinkovitosti *shadera*, što je ranije objašnjeno).

Kod pakiranja parametara se koristi ideja da znamo kolike su granice za pojedine cjelobrojne vrijednosti te u jedan tip podataka možemo spremiti više različitih vrijednosti umjesto samo jednu korištenjem operacija nad bitovima (kao što su *AND*, *OR* ili *SHIFT*). Npr. ako koristimo *uint* (pozitivni cjelobrojni) tip podataka koji je veličine *32 bita* i imamo neke podatke za koje znamo da su uvijek u rasponu *[0, 15]* (koriste maksimalno *4 bita*), onda možemo u jednu *uint* varijablu staviti *8* takvih podataka. Iako se za pakiranje i raspakiravanje koriste dodatne operacije nad bitovima, konačan rezultat je ipak brži nego korištenje velikog broja argumenata u *geometry shaderu*.

Kako su sada objašnjeni razlozi i trikovi zašto koristiti više prolaza moguće je detaljnije opisati što se događa u tim prolazima te kako je *MC* algoritam preveden u kod izvediv na grafičkoj kartici.

7.2.2.1. Prvi prolaz

Kao što je ranije spomenuto, prvo se ulazi u program za sjenčanje vrhova (*Vertex shader*) kojem je ulazni parametar položaj svake kocke. Zatim se taj položaj koristi kao referentna točka za čitanje vrijednosti osam vrhova kocke iz volumnog modela koji je učitan kao *3D* tekstura:

```
cubeV0123.x = densityVolume.Load(Pos + mcSize.yyyy);
            ...
cubeV4567.w = densityVolume.Load(Pos + mcSize.xyxy);
```

Kako performanse programa za sjenčanje ovise o broju instrukcija, a time i varijabli onda je poželjno što češće koristiti optimizirane strukture, tako da se u ovom slučaju vrijednosti ne spremaju u 8 odvojenih varijabli nego u 2 vektora veličine 4 tipa *float*, a varijabla

mcSize kojom su definirani smjerovi vrhova kocke za učitavanje je *2D* vektor gdje je *x = 1*, a *y = 0*, a operacija miješanja vektora (engl. *swizzling*) je besplatna što se tiče računalnih resursa te je puno bolje rješenje od stvaranja 8 različitih *4D* vektora za svaki smjer.

Zatim se te uzorkovane vrijednosti svode na 0 ili 1 kako bi se mogao formirati indeks za tablicu s vrhovima (pretpostavka je da pozitivne vrijednosti predstavljaju puna područja modela te one postaju 1, a negativne vrijednosti su prazne i one postaju 0). To obavlja operacija zasićenja (engl. *saturate*), a vrijednosti se množe s velikim brojem da vrijednosti između 0 i 1 ne bi bile zaokružene na 0 prilikom pretvaranja iz *float* u *int*.

```
uint4 i0123 = (uint4)saturate(cubeV0123*9999);
uint4 i4567 = (uint4)saturate(cubeV4567*9999);
uint cubeCase = (i0123.x     ) | (i0123.y << 1) | (i0123.z << 2) | (i0123.w << 3) |
                (i4567.x << 4) | (i4567.y << 5) | (i4567.z << 6) | (i4567.w << 7);
```

Nakon toga se indeks kocke i njen položaj šalju u program za sjenčanje geometrije (*geometry shader*) gdje se dohvaćaju bridovi iz tablice i obavlja odbacivanje kocaka u slučaju da su potpuno prazne (indeks je 0) ili ako su pune (indeks je 255). Zbog korištenja pakiranja vrijednosti potrebno je podijeliti cijeli volumni model nepoznate veličine u blokove točno određene veličine. U ovom slučaju je odabrana veličina bloka 64 jer je *uint* tip predstavljen s 32 bita, a 64 vrijednosti je moguće predstaviti s 6 bitova, tako da je u jedan *uint* moguće staviti 3 vrijednosti za položaj kocke u *3D* prostoru (3 * 6 = 18) i 3 točke poligona koje su predstavljene indeksima bridova (kocka ima 12 bridova, a za prikaz broja 12 su potrebna minimalno 4 bita, što za 3 točke iznosi 3*4 = 12 bitova). Na kraju je u taj jedan *uint* upakirano 6 različitih vrijednosti s popunjenošću 18 + 12 = 30 od 32 bita. Kako ova vrijednost položaja samo predstavlja lokaciju unutar bloka potrebno je spremiti i vrijednost koji je to blok po redu, odnosno gdje se nalazi u prostoru, a osim redoslijeda bloka još treba spremiti i indeks kocke, tako da se opet obavlja pakiranje, 3 vrijednosti od 8 bitova za položaj bloka i jedan indeks kocke, također veličine 8 bitova. Time je teoretski ograničena veličina volumnog modela na 256 blokova s veličinom 64, što iznosi 16 384 ćelije, no za praktične potrebe je to sasvim dovoljna veličina. Pakiranje se djelomično obavlja u programu za sjenčanje vrhova:

```
uint3 blockNum = voxPos / 64;
uint blockNum_x_y_z_cubeCase = ((0xFF & blockNum.x) << 24) |
                               ((0xFF & blockNum.y) << 16) |
                               ((0xFF & blockNum.z) << 8)  | cubeCase;
uint3 voxPosInBlock = voxPos % 64;
uint voxPos_x6_y6_z6 = ((0x3F & voxPosInBlock.x) << 24) |
                       ((0x3F & voxPosInBlock.y) << 18) |
                       ((0x3F & voxPosInBlock.z) << 12);
```

A ostale vrijednosti (indeksi bridova) pakiraju se u programu za sjenčanje geometrije čiji će isječak koda biti prikazan sljedeći. Navedena petlja ide od 0 do 5 jer je ranije spomenuto da je moguće stvoriti do 5 poligona po kocki:

```
if ((cubeCase != 0) || (cubeCase != 255))
     for (uint i=0; i<5; i++) {
            uint3 triEdgeIntersect = triTable[cubeCase][i];
            if (triEdgeIntersect.x != -1) {
              output.x6_y6_z6_edge1_edge2_edge3 = inp.voxPos_x6_y6_z6 |
                   ((0xF & triEdgeIntersect.x) << 8) |
                   ((0xF & triEdgeIntersect.y) << 4) |
                    (0xF & triEdgeIntersect.z);
              pStream.Append(output);
            }
            else break;
     }
```

U prvom prolazu ne postoji program za sjenčanje slikovnih elemenata (*pixel shader*) jer se dobiveni markeri za trokute ne iscrtavaju izravno nego ih je potrebno još obraditi u drugom prolazu. Stoga se koristi *Stream Output* faza da se ti podaci proslijede u *Stream Output* spremnik. Da bi grafička kartica znala što treba učiniti potrebno je pravilno opisati definiciju prolaza s parametrima koje će sadržavati kao izlaz:

```
pass list_triangles
{
   Profile = 10.0;
   StreamOutputRasterizedStream = 0;
   VertexShader = VS;
   GeometryShader = GS;
   StreamOutput = "TEX0.x; TEX1.x";
   PixelShader = NULL;
}
```

A taj se spremnik zatim može koristiti kao spremnik vrhova za ulaz u program za sjenčanje vrhova (*vertex shader*) u drugom prolazu, a povezivanje se obavi u glavnom programu:

```
GraphicsDevice.SetVertexBuffer(0, soBuffer, 8);
GraphicsDevice.SetVertexInputLayout(soInputLayout);
voxelEffect.CurrentTechnique.Passes[1].Apply();
GraphicsDevice.DrawAuto(PrimitiveType.PointList);
```

Metoda *DrawAuto* koristi se jer nama nije poznat točan broj elemenata koje može vratiti grafička kartica, ali zato ona interno zna koliki je taj broj te ga automatski iskoristi. Naravno, moguće je koristiti i metodu kojoj se kao parametar može dati broj elemenata s koliko je definirana veličina spremnika vrhova, ali tada pate performanse jer se posao mora obavljati i za prazne ćelije koji imaju vrijednosti 0.

7.2.2.1. Drugi prolaz

Drugi prolaz učitava pakirane parametre iz prvog prolaza, zatim ih raspakirava te računa stvarni položaj ćelije u volumnom modelu iz podataka o položaju bloka i položaja ćelije u bloku:

```
uint b = blockNum_x8_y8_z8_cubeCase8;
uint3 blockNum = uint3((b >> 24) & 0xFF, (b >> 16) & 0xFF, (b >> 8) & 0xFF);
uint vp = x6_y6_z6_edge1_edge2_edge3;
uint3 voxPos = uint3((vp >> 24) & 0x3F, (vp >> 18) & 0x3F, (vp >> 12) & 0x3F);
uint edges[3] = { (vp >> 8) & 0xF, (vp >> 4) & 0xF, (vp & 0xF) };
uint3 wsVoxPos = voxPos + blockNum * 64;
```

Ti podaci se zatim koriste za izračun položaja vrha poligona i njegove normale (rezultati iz funkcije *PlaceVertOnEdge* su dobiveni u prostoru modela i trebaju se množenjem s odgovarajućim matricama transformirati u željeni oblik):

```
for (int i = 0; i < 3; i++)
{
     outV = PlaceVertOnEdge(wsVoxPos, edges[i]);
     output.vertPos[i] =  float4(outV.wsCoord, 1);
     output.vertPos[i] = mul( output.vertPos[i], World );
     output.vertPos[i] = mul( output.vertPos[i], View );
     output.vertPos[i] = mul( output.vertPos[i], Projection );
     output.vertNormal[i] = mul( outV.wsNormal, World);
}
```

U funkciji *PlaceVertOnEdge* ponovno se uzorkuju vrijednosti iz volumnog modela te se linearnom interpolacijom na temelju vrijednosti iz dva vrha kocke odlučuje gdje će između njih biti smješten vrh poligona. E*dge_start* i *edge_end* su tablice s predefiniranim

vrijednostima pomoću kojih se na temelju indeksa iz markera određuje gdje se nalaze vrhovi u kocki koji spajaju taj brid:

```
uint4 v0 = uint4(voxPos + edge_start[edgeNum], 0);
uint4 v1 = uint4(voxPos + edge_end  [edgeNum], 0);
float str0 = densityVolume.Load(v0);
float str1 = densityVolume.Load(v1);
float t = saturate( str0 / (str0 - str1) );
float3 vPos = lerp(v0, v1, t).xyz;
```

U toj istoj funkciji računa se i gradijent, a time i normala pomoću novodobivenih točnih vrijednosti položaja vrha poligona. Ovdje se za učitavanje vrijednosti iz volumnog modela koristi metoda *SampleLevel*, za razliku od prethodno korištene metode *Load* jer se sada kao argumenti koriste *float* vrijednosti.

```
grad.x = densityVolume.SampleLevel(samLinearClamp, (vPos + mcSize.xyy) * dimTexInv, 0).x
  - densityVolume.SampleLevel(samLinearClamp, (vPos - mcSize.xyy) * dimTexInv, 0).x;
grad.y = densityVolume.SampleLevel(samLinearClamp, (vPos + mcSize.yxy) * dimTexInv, 0).x
  - densityVolume.SampleLevel(samLinearClamp, (vPos - mcSize.yxy) * dimTexInv, 0).x;
grad.z = densityVolume.SampleLevel(samLinearClamp, (vPos + mcSize.yyx) * dimTexInv, 0).x
  - densityVolume.SampleLevel(samLinearClamp, (vPos - mcSize.yyx) * dimTexInv, 0).x;
output.wsNormal = -normalize(grad);
```

Uzorkovanje na pozicijama koje su različite od cjelobrojnih moguće je jer se prilikom stvaranja *3D* teksture automatski napravi tri-linearna interpolacija te cijeli prostor volumnog modela postaje kontinuiran.

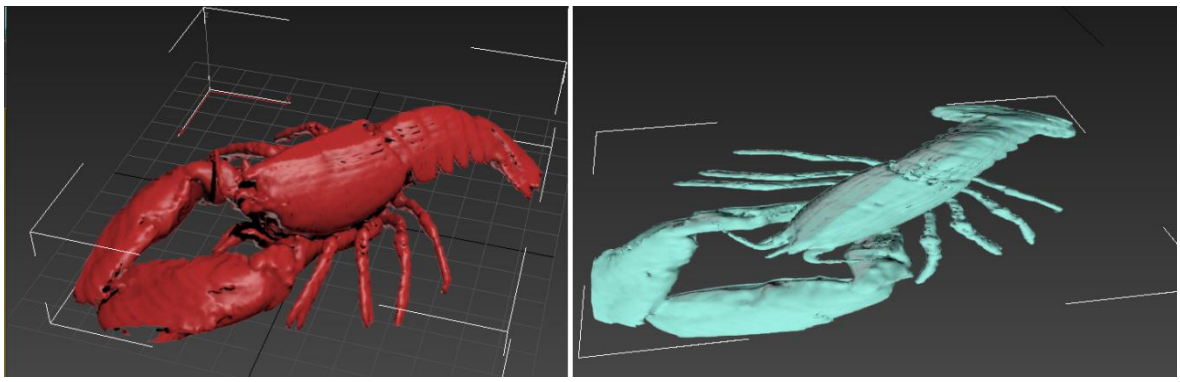
Nakon što su svi podaci izračunati preostaje samo od točaka stvoriti trokute u programu za sjenčanje geometrije (*geometry shader*) na način da se nakon svaka tri poslana vrha u izlazni tok podataka pozove metoda *RestartStrip()*. Svaki izlazni vrh sadrži podatak o položaju transformiran u prostor projekcije i podatak o normali transformiranoj u prostor svijeta. Ta normala se zatim nakon normalizacije koristi za izračun osvijetljenja u programu za sjenčanje slikovnih elemenata (*pixel shader*). Izlaz iz ovog programa su boje kojima se konačna scena iscrtava na ekran i time završava izvođenje *MC* algoritma:

```
float3 outColor = redCol * lightAmbi +
    saturate(dot(lightDir, input.wsNormal) * lightDiff * redCol);
```

Korišten je jednostavan *Phongov* model osvijetljenja, samo s ambijentalnom i difuznom komponentom jer naglasak nije bio na vizualnom rezultatu, nego na funkcionalnom algoritmu *pokretnih kocki*.

7.3. Spremanje modela

Sljedeća je na redu za opisivanje implementacija spremanja modela jer je direktno povezana s drugim prolazom Efekta korištenog za algoritam *pokretnih kocki*. Kao što se u prvom prolazu Efekta koristi *Stream Output* faza da se stvorena geometrija proslijedi u drugi prolaz tako se geometrija (položaji vrhova poligona i pripadajuće normale) iz drugog prolaza šalje u spremnik u glavnom programu. U glavnom programu se zatim stvara datoteka te se u nju upisuju podaci prema pravilima *.obj* formata. Važna stvar na koju treba pripaziti je da su dobiveni podaci već transformirani u prostor projekcije da bi bili pravilno iscrtani na ekranu te stoga model ispada plošan, zato je potrebno svaki vrh prije spremanja pomnožiti s inverzom umnoška matrica svijeta, pogleda i projekcije (Slika 13).



Slika 13 Primjer ispravnog i neispravnog spremanja modela

7.4. Spajanje Kinecta

Zbog korištenja službenog razvojnog seta alata (engl. *SDK, Software Developer Kit*) instalacija i povezivanje vrlo su jednostavni. Upravljački programi dolaze u samoinstalacijskoj arhivi i nakon instalacije dovoljno je spojiti Kinect u računalo, a za komunikaciju programa s Kinectom dovoljno je samo dodati *Microsoft.Kinect.dll*

dinamičku biblioteku kao referencu u izvornom kodu. Iako je moguće odmah nakon dodavanja reference upravljati Kinectom i dohvaćati podatke, Microsoft preporuča korištenje svoje klase *KinectChooser.cs* (iz razvojnog seta alata) za početno postavljanje veze jer ta klasa olakšava kontrolu ako je u sustav spojeno više Kinectova te pravilno postupa ako se program pokrene, a Kinect nije spojen ili ako se prilikom rada odspoji. Iako je ta klasa pisana za korištenje u *XNA Game Studio* razvojnoj okolini, prilagodba za korištenje u *SharpDX-u* je vrlo jednostavna jer je on pisan po uzoru na *XNA* i većina naziva metoda zajedno s parametrima je identična. Od mnogih različitih vrsta podataka koje je moguće dobiti od Kinecta za ovaj program zanimljivi su bili samo dubinska mapa (engl. *depthstream*) i kostur, odnosno položaji šaka.

7.5. Vizualizacija dubinske mape

Prije početka rada na kontroli pomoću ruku trebalo je razviti vizualnu povratnu vezu (engl. *feedback*) koja bi pomogla u lakšem ispravljanju grešaka tijekom razvoja algoritma detekcije prstiju, a kasnije kao pomoć prilikom pozicioniranja osobe ispred Kinecta. Ta vizualna veza ostvarena je u obliku malog potprozora gdje se prikazuju sirovi podaci dubinske mape obogaćeni s mnoštvom drugih informacija.

Vizualizacijski potprozor izveden je kao Efekt i sastoji se samo od programa za sjenčanje slikovnih elemenata (*pixel shader*). Za iscrtavanje se koristi metoda *DrawQuad* koja crta pravokutnik preko cijelog ekrana i automatski kao ulazne parametre programa za sjenčanje stvara vrijednosti u rasponu [0, 1] u *x* i *y* smjeru koje odgovaraju *u* i *v* parametrima za mapiranje teksture. Iako se inicijalno rezultat iscrtava preko cijelog ekrana, lako mu je promijeniti izgled korištenjem standardnih matrica za transformacije kao što su ovdje korištene za skaliranje i translaciju.

```
upperRight = Matrix.Scaling(0.4f) * Matrix.Translation(0.6f, 0.6f, 0);
                                ...
GraphicsDevice.DrawQuad(depthStreamEffect, upperRight);
```

No Efekt ne zna direktno raditi s dubinskim podacima. Stoga je potrebno u svakoj iteraciji petlje za iscrtavanje dohvatiti dubinsku sliku (*frame*), od nje stvoriti *2D* teksturu te je onda poslati u Efekt.

```
using (DepthImageFrame depthImageFrame = kinectSensor.DepthStream.OpenNextFrame(0))
{
  depthPixelData = new short[depthImageFrame.PixelDataLength];
  depthImageFrame.CopyPixelDataTo(depthPixelData);
  depthTexture = Texture2D.New<short>(GraphicsDevice, dsWidth, dsHeight,
      PixelFormat.R16.SInt, depthPixelData);
  depthStreamEffect.Parameters["depthImage"].SetResource<Texture2D>(depthTexture);
}
```

Podatak o dubini zapravo sadrži dvije vrijednosti. Jedna od njih je indeks osobe ispred Kinecta, a druga je zapravo podatak o dubini. Zato ih je potrebno pravilno raspakirati te zatim pretvoriti u *float* vrijednosti da se mogu koristiti kao parametar za određivanje konačne boje slikovnog elementa (*pixel*).

```
int depthPixel = depthImage.Load(realCoo);
int plIdx = (depthPixel & 0x7) % 8;
float depth = (depthPixel >> 3) * maxDepthInv;
```

Taj posao se obavlja u programu za sjenčanje, a osim ovih vrijednosti moguće je učitati i druge podatke, kao što su položaji i dubine ruku i prstiju te tekstura s konturama ruku, no o tome će biti riječi u sljedećem odlomku.

7.6. Detekcija prstiju

Kao što je ranije opisano, detekcija prstiju temelji se na određivanju konture ruke te izvođenja algoritma *k-zakrivljenosti* nad tom konturom. Da bi se stvari ubrzale, traženje konture se ne izvodi preko cijele dubinske mape nego u pravokutniku koji otprilike odgovara veličini otvorene ruke (u određivanju veličine ovog pravokutnika puno je pomogla vizualizacija dubina opisana u prethodnom odlomku), ali da bi znali gdje početi pretraživati potrebne su nam lokacije ruku za što se koriste podaci iz kostura koje dobivamo od Kinecta.

```
foreach (Joint j in skeleton.Joints)
{
    switch (j.JointType.ToString())
    {
        case "HandLeft":
            HandData.jointHandL = j;
            HandData.positionHandL = j.Position.ToVector3wOffset();
            HandData.iPositionHandL = HandData.positionHandL.ToInt3();
            numHands++;
            break;
```

Ovdje je vidljiv primjer za lijevu ruku, a za desnu je postupak isti, samo je drugačiji naziv zgloba. Kako znamo da osoba ima dvije ruke, postupak pretraživanja se prekida kada *numHands* varijabla naraste na 2.

Problem je što kostur sadrži položaje u *3D* prostoru, a nama su potrebne dubine koje se kasnije koriste prilikom određivanja pripada li dio slike ruci, tijelu ili pozadini. Srećom, u *SDK-u* postoji metoda za pravilno mapiranje iz *3D* prostora u *2D* „dubinski“ prostor.

```
DepthImagePoint hL = kinectSensor.CoordinateMapper.MapSkeletonPointToDepthPoint(
    HandData.jointHandL.Position,
    DepthImageFormat.Resolution640x480Fps30);
```

U početku je cijeli algoritam detekcije prstiju izveden samo za jednu ruku, a kada je utvrđena njegova ispravnost refaktoriran je da se može iskoristiti na isti način i za drugu ruku, samo s drugim parametrom lokacije ruke.

Prvi korak algoritma je nalaženje konture ruke. Obavlja se tako da se pravokutnik oko ruke skenira red po red te svaki put kad nađe prijelaz između ruke i pozadine i obrnuto to zabilježi u *2D* polju konture s dimenzijama kao i orginalna dubinska slika (u početku je cijelo polje ispunjeno nulama, a kad se nađe prijelaz, stavi se jedinica). Određivanje pripada li dubinska točka ruci ili ne radi se na temelju uspredbe dubina s poznatom točkom ruke koju smo dobili iz kostura, uz dodatne uvjete, jer se zbog tehnologije kojom je izveden Kinect događaju sjene u kojima je podatak o dubini nepoznat. Kontura se određuje na način da se uvijek gledaju dvije susjedne točke te se zaključak izvodi na temelju prijelaza među njima između ruke i pozadine.

```
for (int y = minY; y < maxY; y++)
    for (int x = minX + 1; x < maxX; x++)
    {
        DepthImagePixel dP = depthPointData[(x - 1) + y * dsWidth];
        if (!dP.IsKnownDepth || (dP.Depth > hand.Depth + offD))
            wasBg = true;
        else
            wasBg = false;

        dP = depthPointData[x + y * dsWidth];
        if (wasBg && dP.IsKnownDepth && (dP.Depth <= hand.Depth + offD))
            contourData[x + y * dsWidth] = 1;
        else if (!wasBg && (!dP.IsKnownDepth || dP.Depth > hand.Depth + offD))
            contourData[(x - 1) + y * dsWidth] = 1;
        else
            continue;
    }
```

Skeniranje se još jednom ponavlja stupac po stupac na analogan način jer bi inače potpuno horizontalne linije koje se pojavljuju u konturi bile preskočene. Nakon što je polje s konturom popunjeno potrebno je stvoriti poredanu listu s položajima točaka koje čine konturu da bi se mogao provesti algoritam *k-zakrivljenosti*.

Stvaranje poredane liste započinje s nalaženjem prve točke konture ruke u prethodno popunjenom polju, a zatim se dalje kružno pretražuju točke udaljene za *1 piksel* u smjeru kazaljke na satu. Svaka nova nađena točka spremljena je u listu, a pretraživanje se pokreće od njene lokacije, ali s početnim smjerom zarotiranim za *-90°* od trenutnog. Takav način pretrage je moguć jer u konturi sigurno ne postoje rupe na mjestima prstiju i najčešće je zatvorena, a ako i bude otvorena, to se dogodi kod zapešća koje nije bitno.

```
for (int i = startDir; i < startDir + 8; i++) {
    int dir = i % 8;
    Int2 dirV = dirVectors[dir];
    x += dirV.X;
    y += dirV.Y;
    if (contourData[x + y * dsWidth] == 1) {
        Vector2 v = new Vector2(x, y);
        contourList.Add(v);
        startDir = dir - 2;
        if (startDir < 0) startDir += 8;
    }
    else {
        x -= dirV.X;
        y -= dirV.Y;
    }
}
```

U prethodnom isječku izvornog koda navedena je samo osnovna ideja algoritma bez dodatnih podešavanja bez kojih ne radi u svim slučajevima. Lista *dirVectors* sadrži vektore s predefiniranih osam smjerova poredanih tako da se povećanjem indeksa za jedan mijenjaju u smjeru kazaljke na satu za 45° (npr. *(-1, 0), (-1, -1), (0, -1), ...*).

Stvorena se lista zatim napokon šalje u funkciju gdje se detektiraju položaji prstiju pomoću algoritma *k-zakrivljenosti*.

```
for (int iIdx = 0; iIdx < cLength; iIdx++) {
    Vector2 p0 = contourList[iIdx];
    int jIdx = (iIdx + kCurv < cLength ? iIdx + kCurv : cLength - 1);
    Vector2 p1 = contourList[jIdx];
    int kIdx = (jIdx + kCurv < cLength ? jIdx + kCurv : cLength - 1);
    Vector2 p2 = contourList[kIdx];
    double angle = getRadAngle(p0, p1, p2);
    if (angle < 0.58) {// rad2deg(0.55) ~= 31.5
        Int2 cent = new Int2((int)(p0.X + p1.X + p2.X) / 3,
                             (int)(p0.Y + p1.Y + p2.Y) / 3);
        DepthImagePixel dP = depthPointData[cent.X + cent.Y * dsWidth];
        //centroid pripada ruci?
        if (dP.PlayerIndex != 0 && (dP.Depth <= hand.Depth + offD)) {
            fingerTips.Add(p1.ToInt2());
            iIdx = (iIdx + 20 < cLength ? iIdx + 20 : cLength - 1);
        }
    }
}
```

Kao što je vidljivo iz isječka, algoritam je vrlo jednostavan i radi na pretpostavci da je cijela šaka blago zakrivljena, a da su vrhovi prstiju pod oštrim kutom (u ovom slučaju oko 30°), a doline između prstiju se izbjegavaju tako da se provjeri pripada li stvarno centroid između te tri točke ruci. Parametar *k* iz naziva algoritma koji određuje koliko će točke biti međusobno razmaknute predstavljen je varijablom *kCurv*.

Dobivene vrijednosti vrhova prstiju predstavljaju točke u *2D* prostoru s njima pripadajućom dubinom, a da bi ih mogli koristiti u programu treba ih transformirati u *3D* prostor scene. Kao što u razvojnom alatu postoji funkcija za pretvorbu zglobova iz kostura u dubinske točke, tako postoji i funkcija koja obavlja obrnutu pretvorbu.

```
DepthImagePoint dip = new DepthImagePoint();
dip.X = fTip.X;
dip.Y = fTip.Y;
dip.Depth = depthPointData[fTip.X + fTip.Y * dsWidth].Depth;
SkeletonPoint fingerSP = kinectSensor.CoordinateMapper.MapDepthPointToSkeletonPoint(
    DepthImageFormat.Resolution640x480Fps30, dip);
```

Postupcima navedenim u ovom poglavlju napokon su dobiveni svi podaci potrebni za uspješno ostvarenje sustava gesti za manipulaciju scenom i modelom.

7.7. Oblikovanje modela i kontrola scene

U prvim je verzijama programa radi testiranja funkcionalnosti bilo implementirano samo oblikovanje modela pomoću šaka dobivenih iz *Kinect* kostura, a njihov položaj je direktno odgovarao položaju unutar kvadra (engl. *Bounding box*) koji omeđuje model. Oko tog položaja bi se zatim modificirala *3D* tekstura koja bi se tako modificirana slala u Efekt. Izmjene na teksturi su izvedene na način da se stvori jednodimenzionalno polje sa željenim vrijednostima (u ovom slučaju su sve iste), a zatim se pomoću klase *ResourceRegion* iz *DirectX-a* oblikuje veličina bloka i umetne na željenu lokaciju u teksturi.

```
float[] subMesh = new float[4 * radius * radius * 2];
for (int i = 0; i < subMesh.Length; i++)
    subMesh[i] = voxVal;
var subBBox = new sdx11.ResourceRegion(iHPos.X - radius, iHPos.Y - radius, iHPos.Z - 1,
                                       iHPos.X + radius, iHPos.Y + radius, iHPos.Z + 1);
testTex.SetData<float>(GraphicsDevice, subMesh, 0, 0, subBBox);
```

Kasnije je došla i mogućnost mijenjanja modela pomoću prstiju. Korišten je isti kod, ali je u pozivu funkcije korišten manji parametar za polumjer (engl. *radius*) s obzirom da su prsti manji nego šake čime je bilo moguće praviti finije izmjene na modelu.

Dodatkom detekcije prstiju došla je i mogućnost korištenja gesti jer je postojalo puno više različitih odnosa među korisnikovim rukama na temelju čega su se mogla izvesti razna stanja koja se mogu asocirati s određenim akcijama za manipulaciju scenom. Cijeli sustav kontrole scene je podijeljen u dva dijela. Jedan koristi tipke na tipkovnici i upravlja kamerom mijenjajući vrijednost matrice transformacije pogleda (engl *View matrix*), a drugi dio kontrole temeljen je na pokretima ruku i gestama koji su mijenjali matricu

transformacije svijeta (engl. *World matrix*), ali za razliku od kamere koja utječe na cijelu scenu, a time i pogled na položaje prstiju, matrica svijeta je bila primijenjena samo na model i odgovarajući kvadar koji ga omeđuje.

Kako je osvježavanje položaja kamere jednostavnije, nije bilo potrebno stvarati posebnu klasu nego su jednostavno uz uvjet za svaku pritisnutu tipku mijenjane vrijednosti rotacije i translacije korištene kao parametar u funkcijama kao što su `Matrix.RotationAxis` koja stvara matricu rotacije oko određene osi ili `Matrix.LookAtLH` za stvaranje matrice pogleda ovisno o položaju promatrača, položaju cilja kamere (engl. *target*) i osi prema gore.

Matrice za manipulacije nad modelom stvarane su u posebnoj statičkoj klasi *HandData* u koju su prethodno spremani podaci o rukama i prstima u raznim fazama programa. U svakoj iteraciji petlje za osvježavanje iscrtavanja na temelju tih podataka, ako su zadovoljeni određeni uvjeti, računaju se matrice translacije, rotacije i skaliranja koje se zatim u glavnom programu množe s matricom svijeta. Uvjeti koji moraju biti zadovoljeni određuju se na temelju stanja šake i prstiju i oni su opisani ranije, u teorijskom opisu značajke za manipulaciju scene, kao što je, da ponovimo, uvjet za skaliranje gdje obje ruke moraju biti izvan modela s jednim prstom ispruženim na svakoj ruci uz njihovo međusobno približavanje ili udaljavanje. Ali kako je ovako stvorena matrica utjecala samo na vizualni prikaz modela (indeksi za određivanje položaja unutar volumne teksture su uvijek fiksni), nije više bilo moguće direktno preslikavati položaje ruku u vrijednosti za modifikaciju volumne teksture, kao što je bio slučaj s verzijom u početnoj fazi razvoja programa. Stoga je svaki put uz matricu za množenje s modelom trebalo stvarati i inverz te matrice kojom su se množili položaji ruku i prstiju, tako da se uvijek vrate na početne položaje, kao da se s modelom ništa nije dogodilo.

Prilikom razvoja gesti, ali i drugih dijelova programa pojavljivalo se mnoštvo problema koji bi u potpunosti zaustavljali napredovanje ili bi stvarali pogrešne vizualne rezultate. Premda je većina uspješno riješena, uvijek su moguća dodatna poboljšanja, no to će biti detaljnije opisano u sljedećim poglavljima.

8. Problemi

Prilikom implementacije verzije programa koja se izvodi na glavnom procesoru nije uglavnom bilo nikakvih programskih problema jer je postojalo mnoštvo dokumentacije i već gotovih projekata sličnog tipa koji su olakšavali razvoj i usput služili kao referentna verzija za usporedbu rezultata. Stoga je, čak i u slučaju da se nešto dogodilo, uvijek bilo moguće pogledati kako je netko drugi pristupio rješavanju problema. No, zato su se događali problemi sa sklopovskom opremom (engl. *hardware*). Na mom računalu nije bilo moguće stvoriti modele veće od dimenzija *210x210x210* jer bi računalo ostalo bez memorije, program bi potrošio oko *1.7 GB RAM-a* i zatim bi se srušio što nije dobro jer mnogi modeli korišteni u praktične svrhe imaju puno veće rezolucije od ovdje navedenih.

Druga verzija, koja se izvodi na grafičkom procesoru, bila je puno kompleksnija od prethodno navedene, i zbog same izvedbe, a i zbog broja implementiranih značajki, a posljedica toga su bili skoro konstantni problemi tijekom razvoja svake od značajki, no najviše ih se dogodilo tijekom upoznavanja s načinom rada programa za sjenčanje geometrije (*geometry shader*) i kasnije izvedbe *MC* algoritma na *GPU*.

Jedan od prvih problema programske prirode koji se pojavio bio je prilikom crtanja omeđujućeg kvadra oko modela, a dogodio se jer su se pozivala dva Efekta zaredom. Jedan je koristio program za sjenčanje geometrije, a drugi nije i ovisno o redoslijedu pozivanja omeđujući kvadar bi nekad bio iscrtan, a nekad ne. Nešto kasnije je otkriveno da je *SharpDX Toolkit* (dodatni set alata koji je korišten i koji olakšava *DirectX* programiranje) još uvijek u razvoju i da je zapravo to ponašanje izazvano greškom u njegovom kodu, a osvježavanjem biblioteka na noviju verziju taj problem je otklonjen.

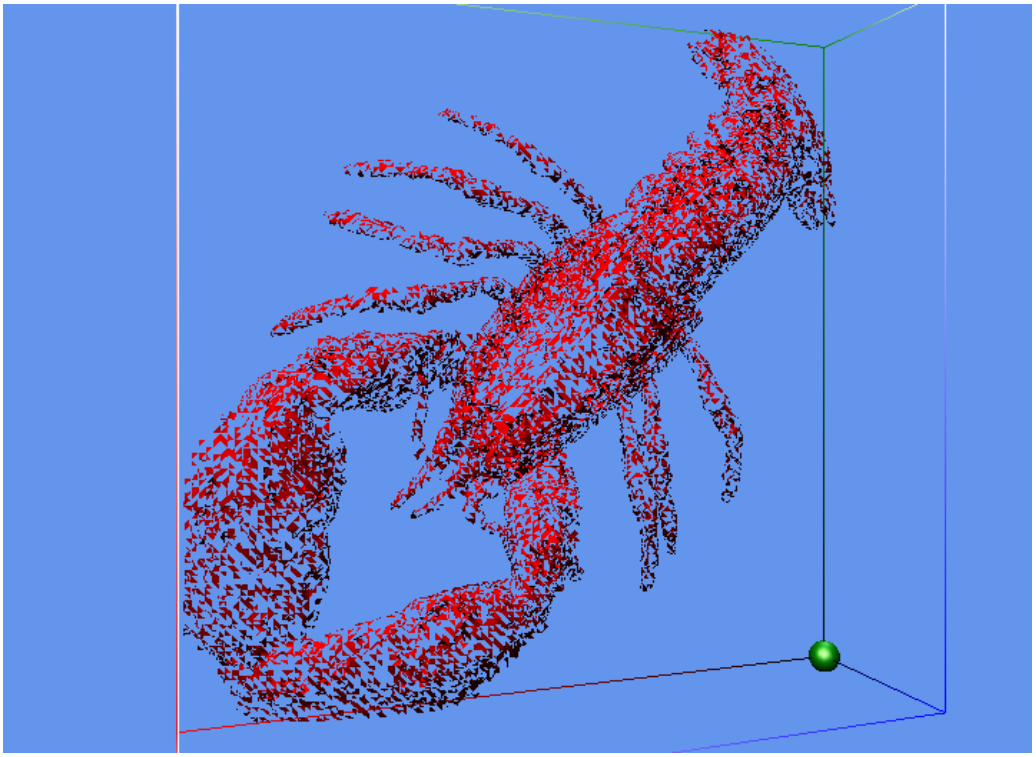
Sličan problem se dogodio i prilikom implementacije *Stream Output* faze. Komandna linija je javljala veliku količinu grešaka vezano za loše parametre predane grafičkoj kartici iako je izvorni kod sigurno bio točan. Takvo ponašanje bilo je sumnjivo i stoga prijavljeno na *SharpDX* razvojnom forumu te se na kraju ispostavilo da *Stream Output* faza uopće nije

bila implementirana unutar *Toolkit* biblioteke. Srećom, *SharpDX* ima dobru korisničku podršku te je za nekoliko dana stigla nova verzija s podrškom za *Stream Output* fazu.

Neki od problema događali su se zbog loše *Microsoftove* dokumentacije za *DirectX* i nedostatka primjera. Jedan od većih koji su spadali u tu kategoriju je korištenje metode za crtanje s krivim parametrom za tip korištenih primitiva, npr.:

```
GraphicsDevice.Draw(PrimitiveType.TriangleList, soBuffer.ElementCount)
```

Očekivano ponašanje bilo je da se koristi onaj parametar koji očekujemo na izlazu jer u slučaju kada se kao ulaz koriste vrhovi, za primitive se navodi lista trokuta. No u slučaju proizvoljnih ulaznih točaka kombiniranih s programom za sjenčanje geometrije koji stvara izlaznu geometriju, trebalo je kao tip primitive navesti listu točaka. U suprotnom se dobiva model pun rupa (Slika 14). Ovaj problem bi se lako izbjegao da je takvo ponašanje negdje dokumentirano, npr. kao opaska u dokumentaciji pokraj liste gdje se navode moguće primitive.



Slika 14 Neispravno iscrtan model pun rupa

Prilikom rješavanja ovog problema puno je pomogao alat za ispravljanje grešaka i grafičku dijagnostiku (*graphics debugger*) ugrađen u *Visual Studio* jer je omogućio praćenje podataka za željene slikovne elemente unutar programa za sjenčanje dok se izvršava, čijom se analizom moglo zaključiti da je s programom za sjenčanje sve u redu i da problem treba tražiti negdje drugdje.

Sljedeći veći problem koji se dogodio zbog dokumentacije bio je prilikom razvoja vizualizacije dubinske mape. Naime, nigdje nije bilo izričito spomenuto na koji su način vrijednosti za indeks i dubinu pakirani u dubinskim točkama. Postojali su neki primjeri programa iz službenog *Microsoftovog* razvojnog seta alata za *Kinect* koji izvlače podatke iz dubinske točke, ali je problem što su svi to radili na drugačiji način. Tako se dobiju prihvatljivi rezultati pomoću kojih se može dobiti vizualizacija dubina, ali je problem što se ne mogu uspoređivati s dubinama iz druge strukture koju je moguće dobiti od Kinecta koja također sadrži dubinske točke, ali se koristi na glavnom procesoru, a usporedba je bila potrebna da se pravilno izdvoje položaji ruku od okoline. Problem je riješen metodom pokušaja i pogreške i uspoređivanjem s vrijednostima iz druge strukture, a kasnije je slučajno nađen članak koji je potvrdio da su prva 3 niža bita indeks korisnika, sljedećih 11 bitova je dubina, a 2 najviša bita se uopće ne koriste.

Posljednja veća skupina problema događala se zbog prirode podataka koji se dobivaju iz Kinecta. Neki podaci sadržavali su mnogo šuma, a nekad bi se događale velike razlike između vremenski susjednih slikovnih okvira (engl. *frame*) ili bi zglobovi bili pogrešno prepoznati što bi ometalo pravilno funkcioniranje korištenih algoritama ili bi se u programu manifestiralo kao trzanje.

To je najviše utjecalo na učinkovitost detekcije prstiju. Da nije bilo šuma ili da su podaci bili zaglađeni i filtrirani, koraci horizontalnog i vertikalnog skeniranja konture mogli bi biti preskočeni i kontura bi se odmah mogla direktno tražiti s algoritmom koji pretražuje točke ukrug. Krive vrijednosti u dubinskim točkama dolazile su do izražaja kod postavljanja uvjeta za detekciju prijelaza između ruke i pozadine jer bi nekad točke ruke koje su imale

postavljen indeks korisnika imale vrijednost dubine koja odgovara dubini pozadine te je trebalo koristiti kompliciranije uvjete nego što bi inače bilo potrebno.

Kod izrade gesti je zbog trzanja trebalo dodavati uvjete pomoću kojih su se ignorirali premali ili preveliki pomaci u određenom vremenu inače je kontrola modela bila praktički nemoguća jer bi model „skakao“ po cijelom ekranu ili bi se tresao, a zbog krive detekcije zglobova bi se izvodio algoritam detekcije na krivom području tijela te bi se prsti naizmjenično pojavljivali i nestajali. Nažalost, neki od ovih problema još uvijek postoje i potrebno je puno vremena za „*naštimavanje*“ uvjeta da manipulacija scene radi uz stabilnu kontrolu, ali zato postoje alternativna rješenja s kojima bi upravljanje bilo bolje.

9. Moguća poboljšanja

Prilikom razvoja svakog programa, nakon njegovog dovršetka uvijek postoji prostor za poboljšanja, a u ovom slučaju ideja ima mnoštvo za svaki dio programa, no bit će navedene samo neke od važnijih koje su izglednije za implementaciju u budućnosti.

Prvo moguće poboljšanje tiče se optimizacije programa za sjenčanje i općenito ubrzanja algoritma pokretnih kocki i izrade programa iskoristivog za praktične svrhe, barem što se tiče računalnih resursa.

Jedan dio ideje je uklanjanje redundantnih vrhova i korištenje indeksiranja jer se veliki dio vrhova dijeli između susjednih bridova kocke i u prosjeku oko pet poligona ima zajednički vrh. Ta optimizacija bi mogla biti izvedena na način da se ne stvaraju vrhovi na svim bridovima za sve kocke, nego samo na 3 brida po kocki, a vrhove na ostalim bridovima bi popunile susjedne kocke, a s dodatnim prolazima u Efektu bi se još jednom prošlo kroz kocke i odredili bi se indeksi vrhova. Druga je ideja vezana za učitavanje volumnog modela. Kako ti modeli imaju vrlo često vrlo visoke rezolucije i izvođenje *MC* algoritma na cijelom modelu bi iznimno usporavalo program moglo bi biti napravljeno dinamičko djelomično učitavanje modela. Naime, model bi bio podijeljen na manje blokove te bi se učitavalo što više blokova najbližih kameri dok god su performanse zadovoljavajuće.

Pomicanjem modela ili kamere učitavali bi se novi blokovi, a stari bi se odbacivali uz spremanje napravljenih promjena.

Iako se dio kontrole programa izvodi pomoću ruku i tipkovnice, radi boljeg korisničkog doživljaja moglo bi biti dodano grafičko sučelje pomoću kojeg bi se omogućilo učitavanje i spremanje modela i mijenjanje određenih parametara bez mijenjanja izvornog koda, ali i prikaz dodatnih informacija o programu kao što su brzina izvođenja, podaci o modelu i slično. Osim lijepog i funkcionalnog sučelja, zgodan grafički efekt bi bilo moguće postići korištenjem položaja glave koji dobijemo iz *Kinecta* za mijenjanje perspektive pogleda čime bi se dobio dojam trodimenzionalnosti na dvodimenzionalnom ekranu.

Kako se veliki dio problema vezanih za kontrolu događao zbog korištenja *Kinecta* za namjenu za koju nije predviđen (detekcija prstiju i komplicirane geste) postoji mogućnost korištenja nekog drugog upravljačkog uređaja, specijaliziranog za kontrolu pomoću prstiju. Jedan od takvih koji je predstavljen ubrzo nakon što je izišao Kinect jest *Leap Motion Controller*. Iako nema mogućnost praćenja drugih zglobova osim prstiju, njihovo praćenje radi odlično i u svim smjerovima, a kako se u ovom programu drugi zglobovi uopće ne koriste, implementacija podrške za *Leap* kontroler ne bi trebala biti veliki problem.

10. Rezultati

S obzirom da dvije varijante programa (*CPU i GPU*) rade na potpuno drugačiji način, nije ih moguće direktno uspoređivati. Stoga će rezultati za različite veličine modela biti prikazani u odvojenim tablicama i grafovima zajedno sa opaskama specifičnima za svaku verziju. Osim numeričkih, na kraju će biti prikazani i vizualni rezultati te njihova usporedba.

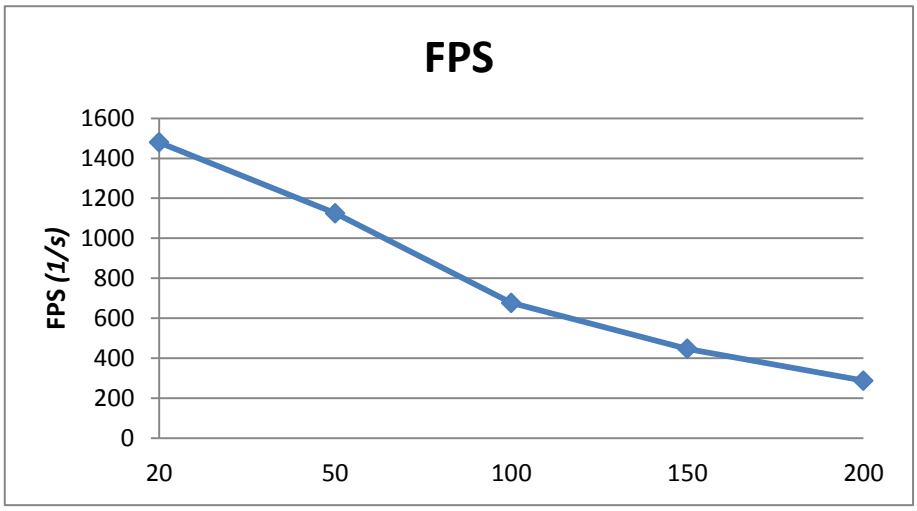
Kako je procesorska verzija prva razvijena njeni rezultati će biti prvi prikazani. Neke od zanimljivih veličina koje je moguće pratiti su brzina iscrtavanja scene, vršna količina potrošene memorije i vrijeme pokretanja, a sve veličine su prikazane u ovisnosti o

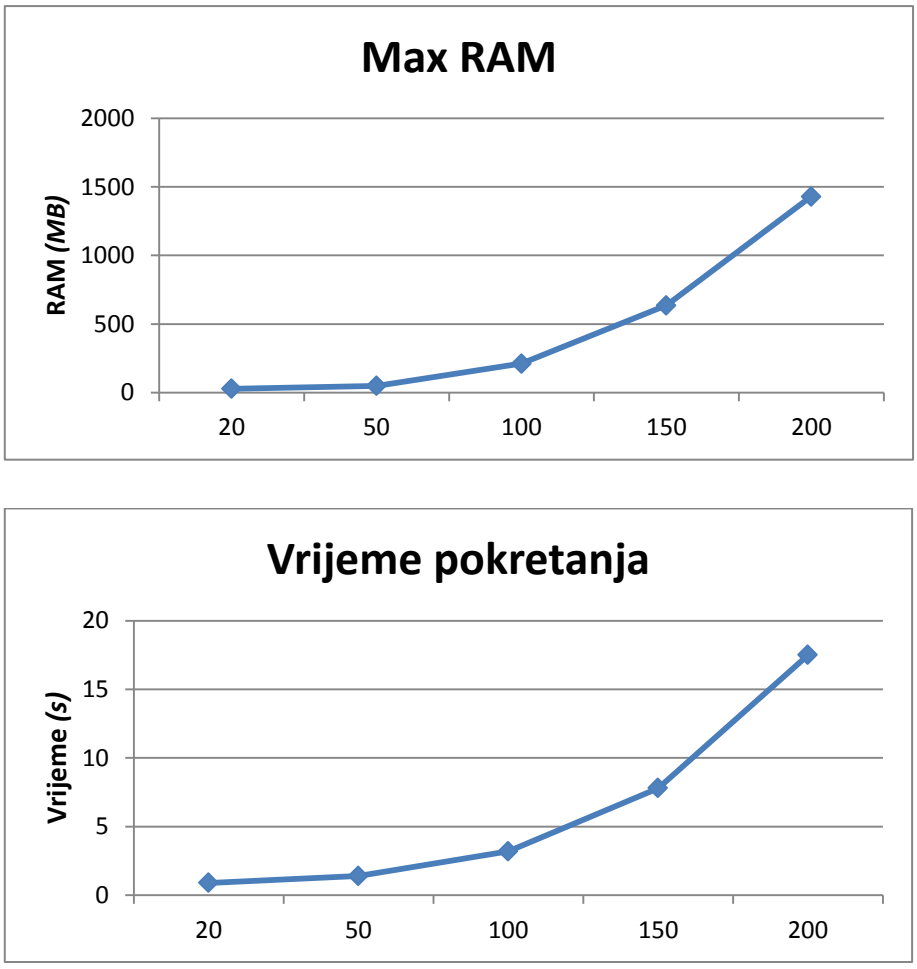
dimenzijama modela. Nisu bila potrebna precizna mjerenja jer su bile dovoljne samo okvirne vrijednosti da se ugrubo vidi složenost algoritama.

Dimenzije su zadavane kao parametar unutar programa te je u ovoj verziji na temelju njih stvaran volumni model. Iako je za njegovo stvaranje bio dostupan *Perlin Simplex* algoritam, korištenje te metode je predugo trajalo te je konačan model bio oblika kocke da se lakše izdvoji vrijeme provedeno u algoritmu pokretnih kocki. Brzina iscrtavanja je mjerena pomoću programa *FRAPS* koji ispisuje broj slika po sekundi sa zanemarivim utjecajem na performanse, potrošnja memorije je mjerena pomoću sistemskog programa *Resource monitor* i uzimana je u obzir samo potrošena fizička radna memorija (virtualna i dijeljena su zanemarene), a vrijeme pokretanja je mjereno pomoću štoperice jer nije bila potrebna velika preciznost.

Tablica 1 Pokretne kocke na *CPU*, veličine ovisne od dimenzijama

Dimenzije modela *(NxNxN)*	Brzina iscrtavanja (FPS) *(1/s)*	Potrošnja memorije (RAM) *(MB)*	Vrijeme pokretanja *(s)*
20	1480	29	0.9
50	1125	50	1.4
100	677	212	3.2
150	447	637	7.8
200	288	1430	17.5





Slika 15 Grafovi rezultata, *CPU* verzija

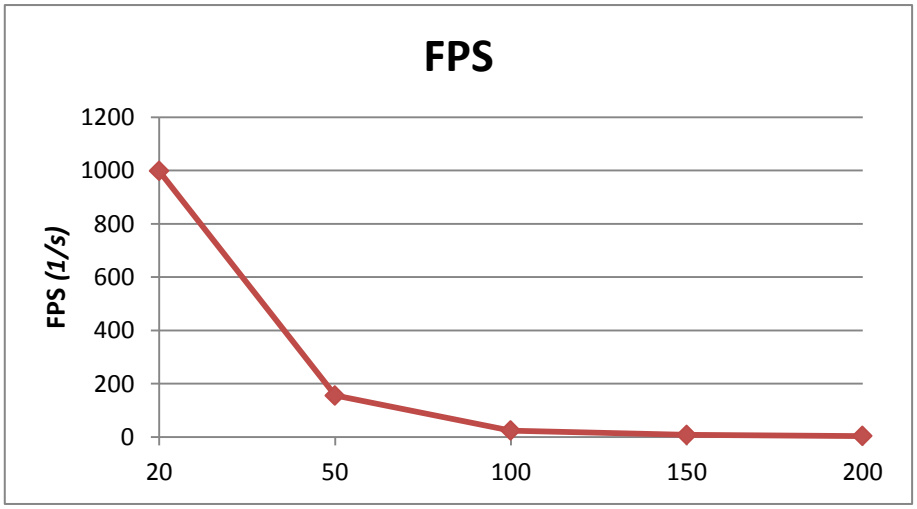
Kao što je vidljivo iz priložene tablice i grafova sve veličine otprilike prate kubnu složenost, što je i bilo očekivano. Kod brzine iscrtavanja je to manje izraženo jer se poligoni ne iscrtavaju u svim kockama (većina ih je prazna). Što se tiče potrošnje memorije, ona se kasnije lako može smanjiti nakon što se popuni spremnik vrhova jer ova verzija nije interaktivna pa sve strukture koje su bile korištene za stvaranje poligona kasnije možemo odbaciti, ali je ipak navedena jer je pokazatelj koliko uopće memorije treba imati računalo da bi se algoritam mogao uspješno izvršiti.

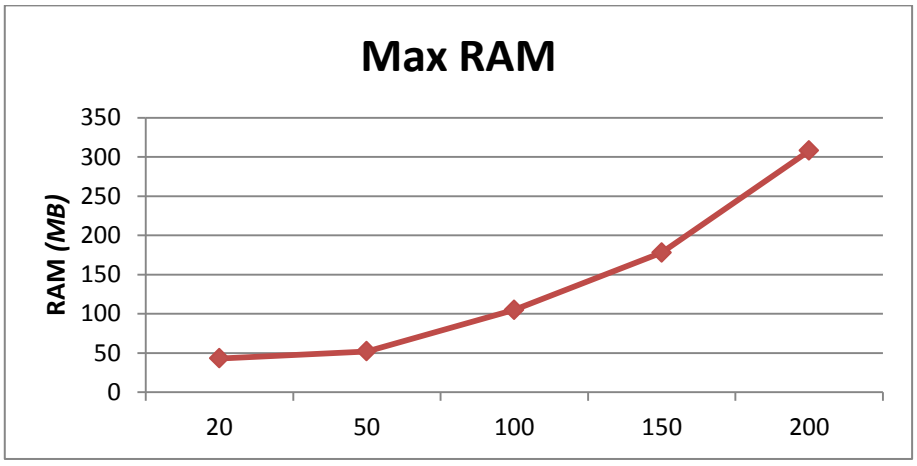
Mjerenja *GPU* algoritma su izvršena na sličan način i s istim alatima, ali je praćeno samo ponašanje algoritma pokretnih kocki (bez vizualizacije dubinske mape i kontrole pomoću

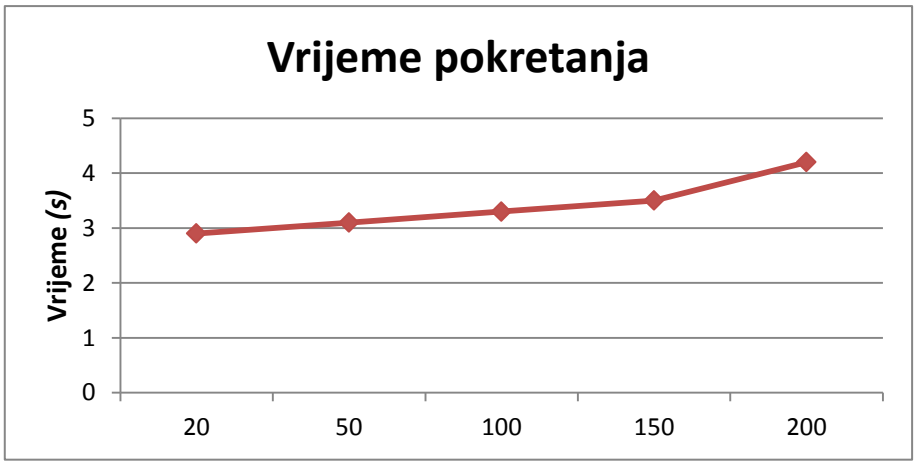
Kinecta) da budu donekle usporedivi s procesorskom verzijom algoritma. A kao model je također iscrtavan kockasti blok s identičnom metodom kao u *CPU* verziji.

Tablica 2 Pokretne kocke na CPU, veličine ovisne od dimenzijama

Dimenzije modela *(NxNxN)*	Brzina iscrtavanja (FPS) *(1/s)*	Potrošnja memorije (RAM) *(MB)*	Vrijeme pokretanja *(s)*
20	998	43	2.9
50	156	52	3.1
100	24	105	3.3
150	8	178	3.5
200	4	308	4.2







Slika 16 Grafovi rezultata, *GPU* verzija

U ovoj verziji iz grafova nije moguće zaključiti puno o složenosti algoritma jer je grafička kartica usko grlo (engl. *bottleneck*) i na visokim rezolucijama se zaguši jer mora obavljati previše posla i s bržom grafičkom karticom dobili bi se mnogo korisniji rezultati. No zato je vidljivo mnoštvo prednosti u odnosu na *CPU* verziju, kao što su iznimno smanjene vrijednosti za količinu potrebne memorije i vrijeme pokretanja. Razlog takvom ponašanju je što se u fazi inicijalizacije stvara jako malo podataka (jedina veća struktura je volumni model), a sve ostalo je zapravo struja podataka koje grafička kartica u svakoj iteraciji direktno stvara, obrađuje i šalje na ekran, bez velike potrebe za spremanjem međurezultata.

Iako su u obje verzije mjerene iste veličine, njihova značenja nisu ista. U *CPU* verziji vrijeme pokretanja uključuje i vrijeme inicijalizacije i vrijeme potrošeno za provedbu algoritma pokretnih kocki, a brzina iscrtavanja samo predstavlja vrijeme potrebno za prikaz stvorenog spremnika vrhova. U *GPU* verziji vrijeme pokretanja predstavlja samo incijalizaciju i slanje podataka na *GPU*, a brzina iscrtavanja uključuje izvršavanje cijelog algoritma pokretnih kocki i prikaz spremnika vrhova.

Na temelju ovih podataka moguće je izdvojiti koliko koja verzija stvarno potroši na izvršavanje jedne iteracije algoritma te usporediti koliko se ubrzanje dobije ovisno o tipu algoritma i korištenog procesora. U ovom mjerenju je za određivanje trajanja korištena

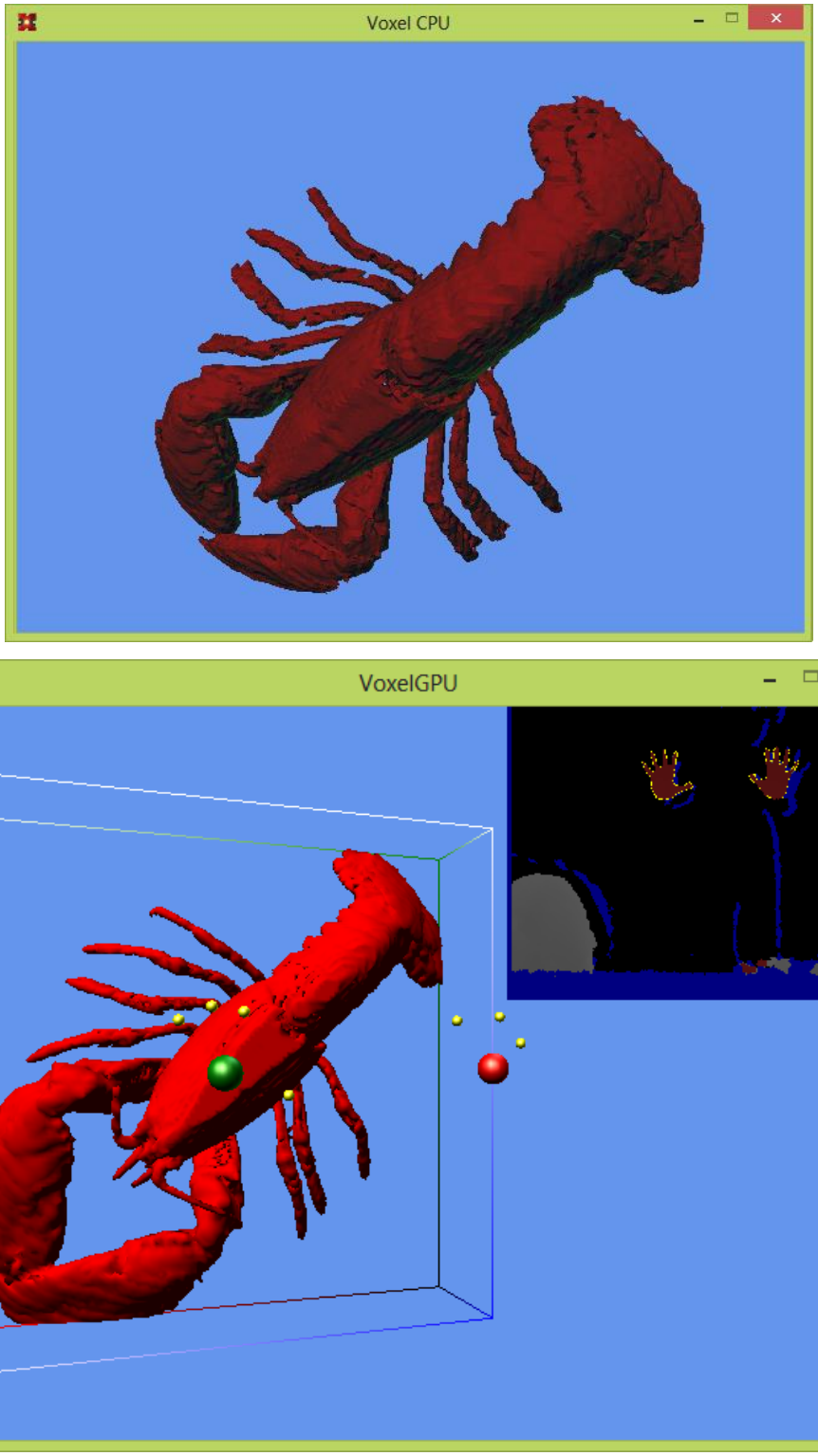
programska štoperica iz *.NET* paketa *System.Diagnostics* jer su bili potrebni precizniji podaci nego u slučaju određivanja okvirnih karakteristika algoritma.

Tablica 3 Usporedba brzine različitih inačica algoritma

Dimenzije modela *(NxNxN)*	*CPU* iteracija *(ms)*	GPU iteracija *(ms)*	Ubrzanje *(CPU/GPU)*
20	34	0.3	113.33
50	291	5.5	52.91
100	1989	40.1	49.60
150	6783	122.7	55.28
200	15125	246.5	61.36

Iz priložene tablice očito je da je u svim slučajevima *GPU* verzija brža od *CPU* verzije, a s rastom rezolucije ta razlika se još povećava. Čudni rezultati na niskim rezolucijama se događaju jer vjerojatno postoje neki dijelovi koda koji se izvršavaju u konstantnom vremenu, a imaju red veličine kao i vremena izvršavanja algoritma pokretnih kocki na niskoj rezoluciji, ali rastom rezolucije ta razlika se jako poveća i do izražaja više dolaze vremena specifična za algoritam. Za usporedbu odnosa između sklopovske i programske opreme (*hardwarea* i *softwarea)*, navodim svojstva procesora korištenih za izračune. *CPU* verzija je koristila *1* jezgru na *2.4 GHz*, a *GPU* verzija se izvršavala na *120* jezgri, svaka takta *0.6 GHz*. Da obje vrste procesora obavljaju jednak broj operacija po taktu, *GPU* verzija bila bi 30 puta brža, ali kako je ta razlika još i veća, znači da je grafički procesor puno učinkovitiji od glavnog procesora za one poslove koje može obavljati.

Što se tiče vizualnih rezultata algoritma, jedina razlika je što *CPU* verzija koristi *konstantno (flat) sjenčanje*, a *GPU* verzija koristi *Gouraudovo sjenčanje*, a što se tiče ostatka programa, *GPU* verzija još prikazuje omeđujući kvadar, dubinsku sliku i položaje ruku i prstiju. Na slikama je prikazan volumni model jastoga učitan iz datoteke. (Slika 17)



Slika 17 Vizualna usporedba *CPU* i *GPU* verzije

11. Zaključak

Kada se *Microsoft Kinect* pojavio na tržištu obećavao je revoluciju u upravljanju računalom, intuitivno, pomoću našeg vlastitog tijela. Nažalost, zbog nedovoljne preciznosti, to se nije dogodilo, ali zato su entuzijasti dobili novu „igračku“ pomoću koje su mogli isprobavati razne nove ideje i algoritme te su računala napokon na jeftin i jednostavan način mogla „vidjeti“ i treću dimenziju.

Kontrola računala pomoću *Kinecta* uz informaciju o trećoj dimenziji omogućila je prototip izvedbe interaktivnog virtualnog modeliranja u prostoru uz direktno korištenje ruku. Osim korištenja *Kinecta*, za uspješnu izvedbu bilo je neophodno odabrati i odgovarajuće algoritme. Kao prikladan algoritam za prikaz i manipulaciju modelom pokazao se algoritam pokretnih kocki (*Marching cubes*) implementiran na grafičkom procesoru zbog dobrog odnosa performansi i vizualnih rezultata. Za detekciju položaja vrhova prstiju korišten je algoritam *k-zakrivljenosti* jer je jednostavan i radi dovoljno dobro za konceptnu verziju programa.

U konačnici se dobiva funkcionalna verzija programa kojoj su najveća mana nespretne kontrole, što bi u budućnosti moglo biti popravljeno zamjenom upravljačkog uređaja. Jedan od potencijalnih kandidata je *Leap Motion Controller* koji je specijaliziran za upravljanje rukama i prstima, a drugi je *Kinect 2* (ako ga bude moguće spojiti na računalo) koji bi općenito trebao biti precizniji od prve inačice.

Literatura

Svi su *internet linkovi* bili provjereni i aktivni 20. lipnja 2013.

[1] Sungmin Cho i ostali. Turn: A Virtual Pottery by Real Spinning Wheel. SIGGRAPH 2012, Los Angeles, California, August 5 – 9, 2012. ISBN 978-1-4503-1435-0/12/0008

[2] Shotton J., Fitzgibbon A., i ostali. Real-Time Human Pose Recognition in Parts from Single Depth Images. Microsoft Research Cambridge & Xbox Incubation.

[3] Liang H., Yuan J., Thalmann D. 3D Fingertip and Palm Tracking in Depth Image Sequences. Nanyang Technological University, Singapore

[4] Jež Karlo. Postupci prikaza terena. Diplomski rad. FER; Zagreb, lipanj 2012.

[5] Persson E. ATI Radeon™ HD 2000 programming guide. AMD Graphics Products Group. Advanced Micro Devices Inc. 2007.

[6] Nvidia. GPU Programming Guide: GeForce 8 and 9 Series. NVIDIA Corporation, Prosinac 2008.

[7] Lorensen W. E., Cline H.E. Marching Cubes: A High Resolution 3d Surface Construction Algorithm. Computer Graphics, Vol. 21, Number 4, Srpanj 1987. str. 164 – 169.

[8] Guennadi Rigue: DirectX10: porting,performance and “gotchas”. Game Developers Conference, San Francisco, Ožujak 2007.

[9] Ryan D. J. Finger and gesture recognition with Microsoft Kinect. Diplomski rad. University of Stavanger

[10] Raheja J. L. i ostali. Tracking of Fingertips and Centers of Palm using KINECT. Machine Vision Lab, Digital Systems Group, CEERI/CSIR, Pilani, Rajasthan, INDIA

[11] Cebenoyan C., Thibieroz N. The A to Z of DX10 Performance. Game Developers Conference, San Francisco, ožujak 2009.

[12] Paul Bourke. Polygonising a scalar field. Svibanj 1994. http://paulbourke.net/geometry/polygonise/

[13] Alexandre Mutel. Benchmarking C#/.Net Direct3D 11 APIs vs native C++. 15. ožujka 2011. http://code4k.blogspot.com/2011/03/benchmarking-cnet-direct3d-11-apis-vs.html

[14] Ryan Geiss. GPU Gems 3: Chapter 1. Generating Complex Procedural Terrains Using the GPU, 2007. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html

[15] Harry Fairhead, All About Kinect. http://www.i-programmer.info/babbages-bag/2003-kinect-the-technology-.html

[16] Lingrand D. The Marching Cubes. http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html

Sažetak

U ovom diplomskom radu opisuje se jedan od mogućih načina izvedbe slobodnog prostornog modeliranja u virtualnom svijetu pomoću direktne manipulacije rukama uz korištenje *Microsoft Kinecta* kao upravljačkog uređaja. Naglasak u radu stavljen je na teorijske osnove korištenih algoritama i bitne dijelove njihove programske implementacije, ali uključeni su i neki od većih problema na koje treba pripaziti, njihova rješenja i moguća buduća poboljšanja. Neki od značajnijih korištenih algoritama uključuju algoritam pokretnih kocki (*Marching Cubes*) za definiranje osobina i stvaranje upotrebljive mase za modeliranje iz volumnih modela, zatim algoritam *k-zakrivljenosti* za prepoznavanje prstiju i raznovrsni algoritmi za detekciju obrisa i ostvarivanje virtualne kontrole pomoću gesti. Konačan rezultat je funkcionalan program koji omogućuje izmjene postojećih ili stvaranje novih modela te njihovo spremanje u datoteku kompatibilnu s klasičnim programima za *3D* modeliranje radi daljnje obrade.

KLJUČNE RIJEČI: Kinect, *3D*, volumno modeliranje, pokretne kocke, Marching Cubes, prepoznavanje, detekcija, prsti, ruke, *SharpDX, DirectX,* Geometry shader, *GPGPU*

Abstract

This thesis describes one of the possible ways for creating freeform spatial modeling in a virtual world using direct hands manipulation with *Microsoft Kinect* as a controller. Emphasis in the thesis is given to theoretical bases of used algorithms and important parts of their programmatic implementations, but it also includes some of the larger issues which were encountered, as well as their solutions and possible future improvements. Some of the more notable algorithms that were used include *Marching Cubes* algorithm for defining properties and creation of usable modeling mass from volumetric models, then *k-curvature* algorithm used for finger recognition and also various algorithms for contour detection and realization of virtual control with gestures. Final result is a functional program that enables the user to change existing or create new models and save them to files compatible with classic *3D* modeling tools for further processing.

KEYWORDS: Kinect, *3D*, volumetric modeling, Marching Cubes, fingers, hands, recognition, detection, *SharpDX, DirectX,* Geometry shader, *GPGPU*